SMC-TR-02-13

AEROSPACE REPORT NO.
TR-2002(8550)-1

Object-Oriented Metrics

20 January 2002

Prepared by

L. A. ABELSON
Software Engineering Subdivision
Computer Systems Division

Prepared for

SPACE AND MISSILE SYSTEMS CENTER
AIR FORCE MATERIEL COMMAND
2430 E. El Segundo Boulevard
Los Angeles Air Force Base, CA 90245

Engineering and Technology Group

THE AEROSPACE CORPORATION
El Segundo, California

This report was submitted by The Aerospace Corporation, El Segundo, CA 90245-4691, under Contract No. F04701-00-C-0009 with the Space and Missile Systems Center, 2430 E. El Segundo Blvd., Los Angeles Air Force Base, CA 90245. It was reviewed and approved for The Aerospace Corporation by M. A. Rich, Principal Director, Software Engineering Subdivision. Michael Zambrana was the project officer for the Mission-Oriented Investigation and Experimentation (MOIE) program.

This report has been reviewed by the Public Affairs Office (PAS) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nationals.

This technical report has been reviewed and is approved for publication. Publication of this report does not constitute Air Force approval of the report's findings or conclusions. It is published only for the exchange and stimulation of ideas.

Michael Zambrana
SMC/AXE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE (*DD-MM-YYYY*)	**2. REPORT TYPE**	**3. DATES COVERED *(From - To)***
20-01-2002		

4. TITLE AND SUBTITLE	
Object-Oriented Metrics	**5a. CONTRACT NUMBER** F04701-00-C-0009
	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)	
L. A. Abelson	**5d. PROJECT NUMBER**
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	**8. PERFORMING ORGANIZATION REPORT NUMBER**
The Aerospace Corporation Computer Systems Division El Segundo, CA 90245-4691	TR-2002(8550)-1

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)	
Space and Missile Systems Center Air Force Space Command 2430 E. El Segundo Blvd. Los Angeles Air Force Base, CA 90245	**10. SPONSOR/MONITOR'S ACRONYM(S)** SMC **11. SPONSOR/MONITOR'S REPORT NUMBER(S)** SMC-TR-02-13

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The application of object-oriented methodology and an evolutionary approach to the development of software-intensive systems introduces some unique acquisition management challenges. This report discusses the unique challenges of using metrics to monitor software developed using object-oriented techniques and evolutionary development lifecycle. Topics addressed include the definition of 31 specific metrics that may be used for monitoring object-oriented design and development. In addition, this report provides guidance on which metrics would be useful during the process of transitioning to the object technology.

15. SUBJECT TERMS

Metrics, Object-Oriented Metrics

16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**	**19a. NAME OF RESPONSIBLE PERSON**
a. REPORT	**b. ABSTRACT**	**c. THIS PAGE**		87	Linda Abelson
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED			**19b. TELEPHONE NUMBER *(include area code)*** (310)336-7350

Acknowledgments

The technical content, as well as the preparation, of this report was developed under the Mission-Oriented Investigation and Experimentation (MOIE) program (Software Acquisition Task). The author wishes to thank the following members of the Computer Systems Division (CSD) and the Reconnaissance Systems Division (RSD) for their review of this report and suggestions for improvements.

- Richard Adams, Senior Engineering Specialist, CSD
- Sergio Alvarado, Director, CSD
- Sheri Benator, Manager, CSD
- Colleen Ellis, Senior Engineering Specialist, CSD
- Suellen Eslinger, Distinguished Engineer, CSD
- Rick Johnson, Engineering Specialist, CSD
- Larry Miller, Senior Engineering Specialist, CSD
- Mary Rich, Principle Director, CSD
- Bonnie Troup, Senior Engineering Specialist, RSD

In addition, the author wishes to acknowledge the contribution *of Appendix D - Acquisition and Development Metrics Planning* by Rita Creel, Senior Engineering Specialist, RSD.

Contents

Preceding Pages Blank

Figures

Tables

1. The Object-Oriented Paradigm

Most engineering disciplines use measurement to manage and control research and development, to monitor progress and the quality of products under development, to gauge the effects of decisions and process changes, and to quantify the impacts of external influences. This has not typically been the case for software engineering. However, the Software Program Managers Network (SPMN) has highlighted as a Principle Best Practice the need for more accurate and timely status information, as well as data that will enable early problem and risk identification. [SPMN] As a result, there is a growing interest in the software engineering community in applying measurement to software systems via the use of software system metrics.

Promises of great improvements in productivity and quality have led many organizations to adopt object-oriented (OO) development methods. These promises have been based on the assumption that significant reductions in development and maintenance costs are possible due to the way in which OO systems are structured. Object-based designs have the potential to reduce the scope and impacts of software changes and to lead to a greater capacity for reuse. Such benefits are very attractive in today's environment of rapidly evolving needs and decreasing budgets. However, for most organizations that have attempted it, the transition from conventional to OO development has been far from trouble free. Often, the results have been disappointing, at times seeming to be a step backward rather than forward. In many cases, unrealistic expectations are partially to blame; insufficient experience with the OO paradigm and lack of familiarity with the specific OO methodology used and associated metrics are also contributing factors.

The purpose of this report is to provide guidance in developing and using metrics for OO systems. This report addresses only metrics specific to OO development. It does not address those metrics common to all developments independent of methodology used. As with conventional development, metrics used in OO development can be helpful not only in providing information on development status, problems, and risks, but also in evaluating the effectiveness of OO methods and tools themselves. Measurement can be used to track progress toward taking full advantage of the OO paradigm. OO metrics can help assess whether or not OO methods are being effectively used to facilitate modifiability and reusability. As organizations modify their OO processes with the intent of improvement, OO metrics are needed to assess whether or not intended improvements are indeed realized.

Coverage of OO metrics for the full life cycle is addressed here; it is not enough to focus on detailed design and code. The most costly errors in software system development originate in the earliest activities and products, particularly in requirement analysis and architectural design. Use of a metrics approach that addresses the progress and quality of these activities and their products can aid early identification of problems and risks and forestall the domino effect of negative impacts to downstream activities and products.

This report documents the unique challenges of developing software using object-oriented techniques, and provides specific measurements and metrics that may be utilized for monitoring the development

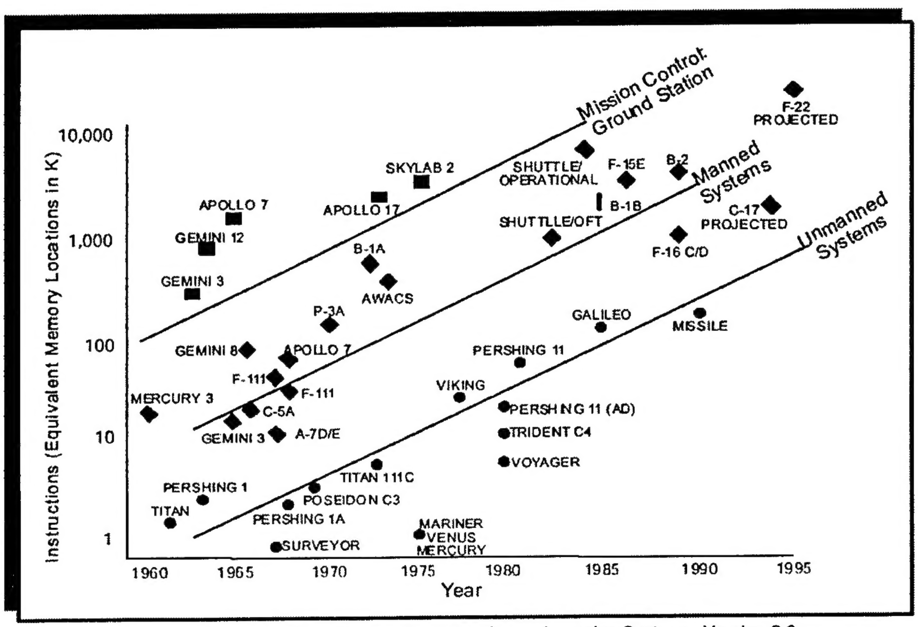
of object-oriented software systems. In addition, this report provides guidance on which metrics would be useful during the technology transition process.

The report consists of five sections and four appendixes. Following this introduction, Section 2 defines *metrics for the object-oriented paradigm* by providing a discussion of the metrics framework, describing the unique attributes of the object-oriented paradigm and providing justification for the selection of OO-unique metrics. Section 3 provides a catalog of object-oriented unique metrics. Section 4 provides guidance on metrics selection. Finally, Section 5 contains conclusions.

The report's four appendixes contain supplementary information. Appendix A provides formulas for calculating metrics presented in Section 3.0, and Appendix B provides a list of references. Appendix C provides background information on OO techniques. Appendix D provides guidance on acquisition and development planning for metrics use.

1.1 Historical Justification for OO

The management of software-intensive systems is relatively new. During the 1960s, the F-4 Phantom used virtually no software in its weapon systems, and software was used sparingly in the DSP satellite. During the 1970s, the rapid evolution of sophisticated electronic circuitry resulted in smaller processors producing more computing power for a fraction of the cost. These advances, compounded by more demanding requirements, dramatically increased DoD's software use. Figure 1.1-1 represents a summary of Air Force and NASA software system size growth between 1960 (Vietnam War) and 1995 (post-Gulf War). [GSAM]



Guidelines fo Successful Acquisition Management of Software Intensive Systems. Version 3.0, Department of the Air Force Software Technology Support Center, May 2000

Figure 1.1-1. Software complexity continues to increase. [GSAM]

Software management technologies have been crucial in keeping up with this explosion in software capacity. Software development productivity measured from the start of development through final qualification test has grown almost linearly from 1960 through the present. A simplified productivity growth curve in Figure 1.1-2 shows this growth. The result shows software development productivity, increasing about one source line per person-month per year over the entire 30-year period. [JENSEN] While no new technology has solved the software productivity problem, object technologies have shown promise in improving the ability of software development organizations to field highly complex software systems.

In order to determine which metrics will be the most useful to the object-oriented developer a complete understanding of how OO techniques differ from traditional techniques is required. A historical justification for the evolution to OO techniques, a comparison between traditional and OO design methodologies, and a discussion on OO lifecycles is provided in Appendix C for this purpose.

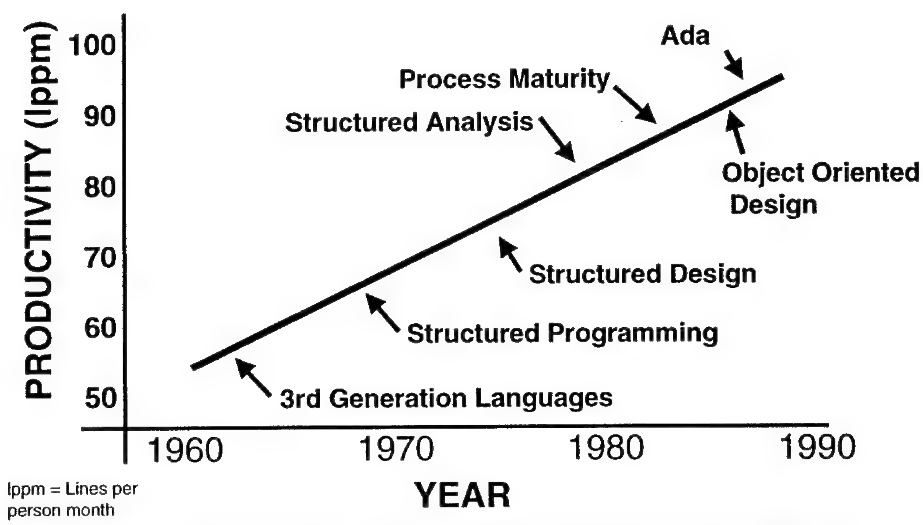
1.2 Definitions of Relevant OO Terminology

The set of definitions included below have been provided based on utilization in this report.

Aggregate. An aggregate is (1) A class that represents the "whole" in an aggregation (whole-part) relationship [OMG]. (2) A numeric value obtained by summing values at lower levels in the aggregation structure.

Ancestor. The ancestors of a class include its parent classes and their ancestor classes.

Association. An association is a semantic relationship between two classes.



Randall W. Jenson, Letter to the Editor, CROSSTALK. Vol. 13, No. 8, August 2000, p. 30.

Figure 1.1-2. Software productivity continues to increase.

Attribute. An attribute is (1) A characteristic of a process, product, or resource. Some attributes can be measured or quantified through observing the product, process, or resource itself, or through observing its behavior. Examples of attributes include size, completeness, volatility, complexity, and traceability. (2) A data item that is encapsulated in a class along with its associated methods. Adapted from Briand, Daly, and Würst [BRIAND].

Child. In a generalization relationship, a child is the specialization of the parent. A child is also called a subclass and is a descendant class of its parent classes.

Class. A class is a structure for OO development that encapsulates data (attributes) and functions (methods). [BRIAND] Design requirements are allocated to classes.

Class Category. A class category is a logical collection of classes, some of which are visible to other class categories and others of which are hidden. The classes in a class category collaborate to provide a set of services. [BOOCH]

Complexity. Cyclomatic complexity is a measure of the complexity of a module's decision structure. It is the number of linearly independent paths and, therefore, the minimum number of paths that should be tested [MCCABE].

Dependency. A dependency is a relationship between two modeling elements in which a change to one element (the independent element) affects the other element (the dependent element).

Descendant. The descendants of a class include its child classes and their descendant classes.

Inheritance. Inheritance is the mechanism by which specific classes incorporate the structure and behavior of more general ancestor classes. Adapted from Object Management Group [OMG].

Method. A method is a procedure or function that is encapsulated in a class with its related data (attributes). Other names for method include Operation, Service, and Member Function. Adapted from Briand, Daly, and Würst [BRIAND].

Object. An object is an instance of a class that has a well-defined boundary and identity that encapsulates state and behavior. Adapted from Object Management Group [OMG].

Object-Oriented Analysis (OOA). Object-oriented analysis is a methodology of analysis in which requirements and potential high-level system structure are examined from the perspective of the classes and objects found in the vocabulary of the problem domain. Adapted from Booch [BOOCH].

Object-Oriented Design (OOD). Object-Oriented Design is a methodology of design in which system design is expressed in terms of classes and objects from the problem domain and relationships between these classes and objects. OOD results in models of the system under design; each model may be depicted by one or more diagrams that show structural, behavioral, and interface characteristics of the system.

Object-Oriented (OO) Development. Object-oriented development is development in which object-oriented analysis, object-oriented design, and object-oriented programming methodologies are used to produce software implementations. Note that some "object-oriented" development efforts may choose to use one or more of OOA, OOD, or OOP, but not others.

Object-Oriented Programming (OOP). Object-oriented programming is an implementation methodology in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships. [BOOCH]

Parent. In a generalization relationship, a parent is a generalization of another element, the child. A parent is also called a superclass and is an ancestor class of its child classes.

Use Case. A description of a set of related sequences of actions (scenarios), including variants, that a system performs that yields an observable result of value to a particular actor (paraphrased from Jacobson, Booch, and Rumbaugh [JACOBSON]). Functional requirements are allocated to use cases.

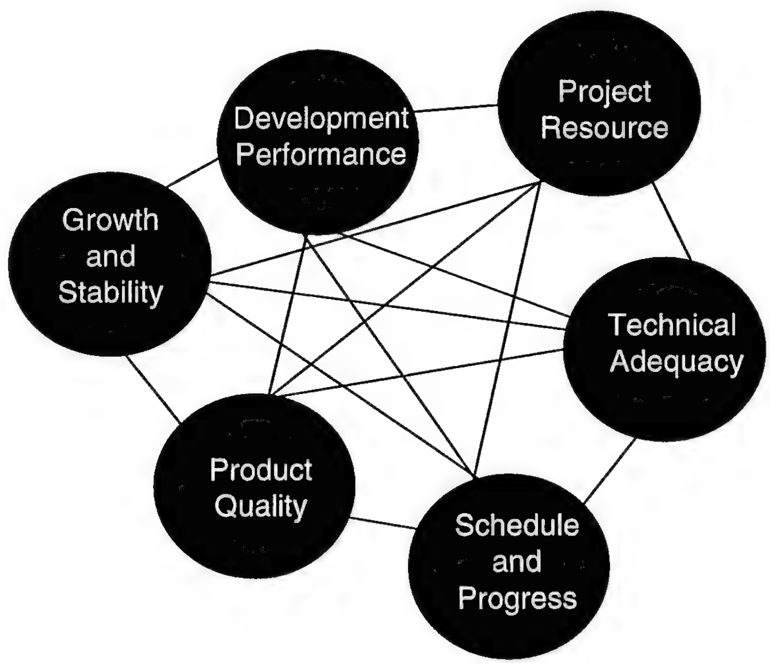
Scenario. A scenario is a specific sequence of actions that illustrates behavior. [JACOBSON] Also, a scenario is a specific instance of a use case.

2. Metrics for Use with Object-Oriented Techniques

2.1 Overview of the Metrics Framework

The information presented in this section is a brief description of the framework described in Section 2.0 (Tailoring Software Measures) of the Practical Software Measurement (PSM). The PSM guidebook is an extensive reference that describes how to define and implement a software measurement process to support the information needs of software-intensive acquisition and development. The PSM guidebook defines a Metrics framework that consists of 6 interrelated software issues. Issues are defined as real or potential obstacles to the achievement of project objectives. These 6 issues are then used to organize specific metrics into categories. This report extends the metric categories and the metrics to include information pertinent to the OO software developer.

The PSM guidebook groups software into six interrelated issues (see Figure 2.1-1) that are common to all projects. These six issues are defined in the "Practical Software Measurement" metrics guidebook and summarized below [PSM].



"Practical Software Measurement", Office of the Under Secretary of Defense for Acquisition and Technology Joint Logistics Commanders Joint Group on Systems Engineering (OUSD A&T JGSE), Version 3.1a, April 1998.

Figure 2.1-1. Interrelationship of metrics issues.

Schedule and Progress – This issue relates to the completion of major milestones and individual work units. A project that falls behind schedule usually can make delivery only by eliminating functionality or sacrificing quality.

Project Resources – This issue relates to the balance between the work to be performed and personnel resources assigned to the project. A project that exceeds the budgeted effort usually can recover only by reducing software functionality or sacrificing quality.

Growth and Stability – This issue relates to the stability of the functionality or capability required of the software. It also relates to the volume of software delivered to provide the required capability. Stability includes changes in scope or quantity. An increase in software size usually requires increasing the applied resources or extending the project schedule.

Product Quality – This issue relates to the ability of the delivered software product to support the user's needs without failure. When poor quality product is developed, the burden of making it work usually falls on the sustaining engineering organization.

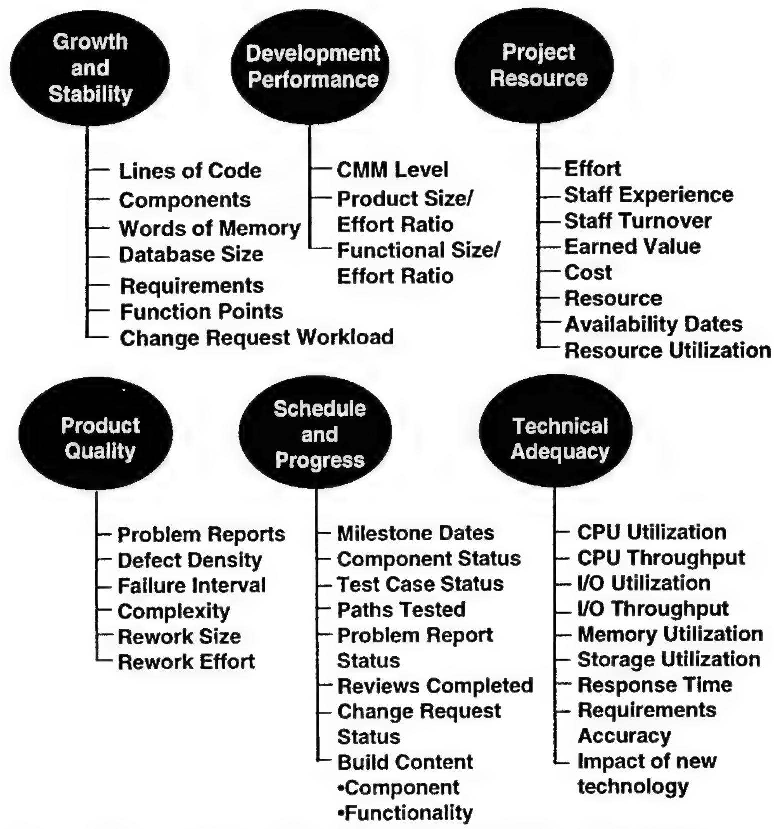
Development Performance – This issue relates to the capability of the developer relative to project needs. A developer with a poor software development process or low productivity may have difficulty meeting aggressive schedule and cost objectives. More capable software developers are better able to deal with project changes.

Technical Adequacy – This issue relates to the viability of the proposed technical approach. It includes features such as software reuse, use of COTS software and components, and reliance on advanced software development processes. Cost increases and schedule delays may result if key elements of the proposed technical approach are not achieved.

These common software issues can be used in two ways. First, the common software issues are used to classify project-specific issues identified via risk analysis or other means so that they can be mapped into the measurement selection structure. Second, reviewing the common software issues helps both the acquirer and the developer to check that all potential issue areas have been considered. [PSM]

Once the project-specific issues have been identified, appropriate measures must be selected to track them. A measure is a quantification of a characteristic or attribute of a software process or product or resource. Many different measures may apply to an issue. However, in most cases it is not practical to collect all or even most of the possible measures for an identified issue. Generally, more measures should be collected to track the high-priority issues. Identification of the "best" set of measures for a project depends on a systematic evaluation of the potential measures with respect to the issues and relevant project characteristics. Section 4.0 of this report on metrics selection provides additional guidance on this topic.

A measurement category is a set of related measures. The measures within a category address related software attributes. They provide similar information and answer similar questions about an issue. A number of measurement categories for each software issue are identified in Figure 2.1-2, which lists the categories for each issue.



"Practical Software Measurement", Office of the Under Secretary of Defense for Acquisition and Technology Joint Logistics Commanders Joint Group on Systems Engineering (OUSD A&T JGSE), Version 3.1a, April 1998.

Figure 2.1-2. Software issue to category mapping.

2.2 Selection and Justification of OO-Unique Metrics

While this report's focus is on OO, it is important to recognize that conventional and OO developments have many features in common. A software development project utilizing the OO design methodology and life cycle has the same software development issues as does a conventional project. The following metrics have been selected for inclusion in this report based on an analysis of the similarities and differences between OO developments and traditional development. The contrast between OO and traditional development is documented in appendix C.

All of the attributes in the three common software issues of Development Performance, Technical Adequacy, and Project Resources remain the same; that is to say, the metrics for these issues are exactly the same in OO as in conventional software development. The categories described in the three common software issues of Growth and Stability, Product Quality, and Schedule and Progress are slightly different. This relationship is summarized in Figure 2.2-1 and can be contrasted with Figure 2.1-1. The issues highlighted in black are those of interest to the OO developer.

A number of measurement categories have been added and modified as a result of the use of OO techniques. These are summarized in Figures 2.2-2, 2.2-3, and 2.2-4. The numbers circled in black on each of these figures uniquely identify the measurement categories that have been modi-

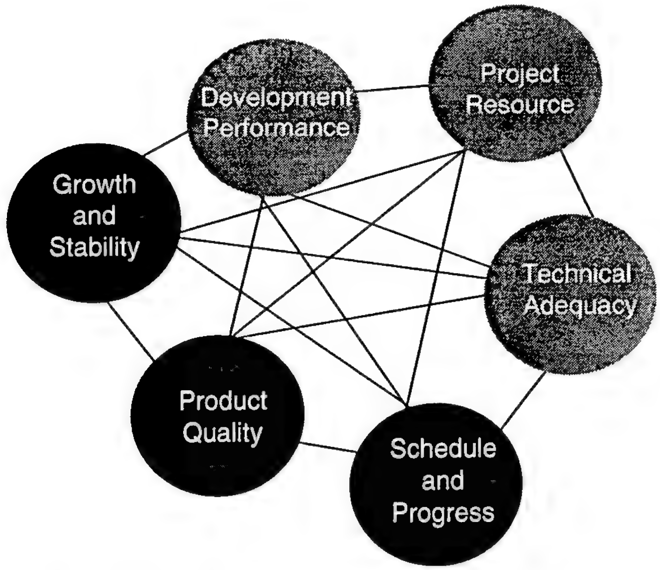


Figure 2.2-1. Interrelationships of OO metrics issues.

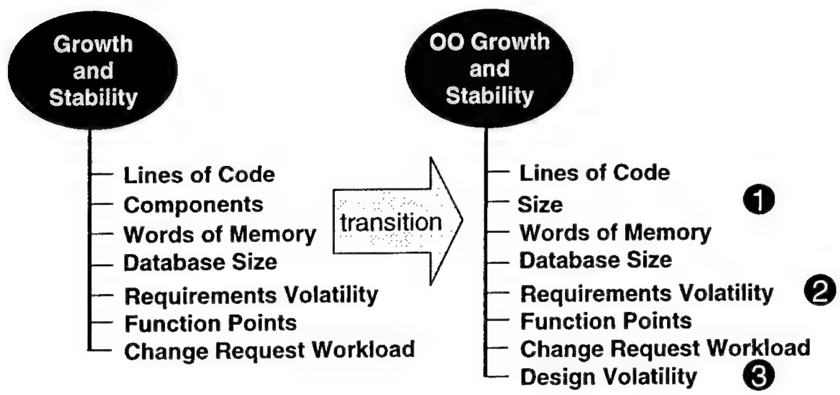


Figure 2.2-2. Software OO issue to category mapping for Growth and Stability.

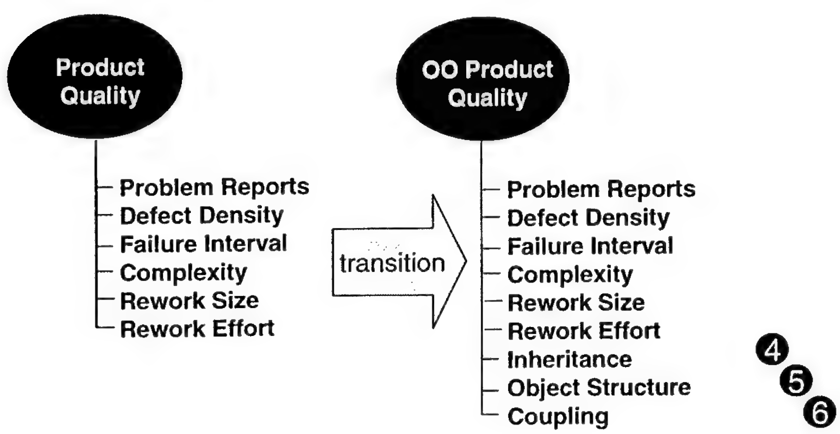


Figure 2.2-3. Software OO issue to category mapping for Product Quality.

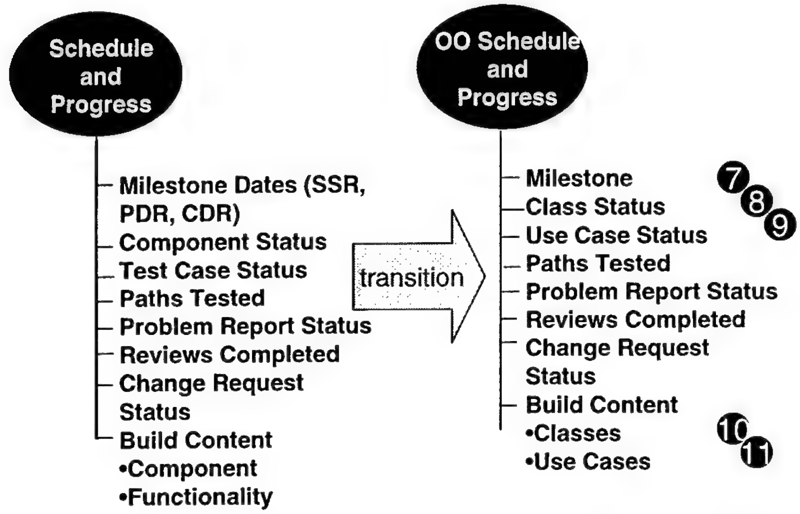


Figure 2.2-4. Software OO issue to category mapping for Schedule and Progress.

fied or added and that are of particular interest to the OO developer. These identifying numbers are one-to-one traceable to the metrics described in Section 3.0, *Catalog of OO Unique Metrics.*

The three measurement categories selected for addition/modification in the Growth and Stability issue are the Size, Requirements Volatility, and Design Volatility measurement categories. Each of these relies on measurement values that are uniquely described in the OO techniques. The size metric, depending on the phase of the program, is defined by use-cases and class definitions. Requirement Volatility is measured in Use-Cases, and Design Volatility is measured in Classes.[1]

The three measurement categories selected for inclusion in the OO Product Quality category are Inheritance, Object Structure, and Coupling. These three measurement categories are new to the metrics framework.[2]

Five measurement categories have been selected for modification and inclusion in the OO Schedule and Progress category. These include: Milestone, Class Status, Use-Case Status, and Build Content – classes and use cases. The modified measurement category is the Milestone metric. In general, this measurement category has been modified to reflect an evolutionary terminology in the milestone titles. The remaining four metrics are new to the metrics framework.[3]

---

[1] The concept of Application Size and Method Size is discussed in LORENZ. ROSENBERG summarizes some of the metrics described in LORENZ, and CHIDAMBER and maps the metrics to a set of quality critiera, including: Classes, Message, Cohesion, Coupling and Inheritance.

[2] Metrics for Class Inheritance, Method and Class Internals, and Class Externals are discussed in LORENZ. CHIDAMBER provides information pertaining to the rationale for the use of the Weighted Methods per Class, Depth of Inheritance Tree, Number of Children, Coupling between Object Classes, Response for Class, and Lack of Cohesion in Methods metrics. ROSENBERG summarizes some of the metrics described in LORENZ and CHIDAMBER and maps the metrics to a set of quality critiera, including: Classes, Message, Cohesion, Coupling, and Inheritance.

[3] Metrics for Scheduling are discussed in LORENZ.

3. Catalog of Recommended OO Unique Metrics

The recommended Object-Oriented Metrics consist of the set of 31 metrics identified in Table 3-1. These metrics were selected by first performing an analysis of the differences between traditional design methodology and Object-Oriented methodologies. These differences were then used to identify common issues in the PSM metrics framework [PSM] where metrics information would be unique to an Object-Oriented development. Once this was accomplished, relevant metrics suitable to fill in the metrics framework were identified based on an industry literature search and the author's experiences. This analysis is summarized in Section 2.0, *Metrics for use with Object-Oriented Techniques.*

The metrics described in the following sections are organized by common issues and metrics categories. This organization is similar to that found in the PSM framework. Each metric description consists of a table describing the metric and an example metric report that indicates how the metric could be represented. The metric table contains information on how the metric fits into the Metrics Framework, a description of the measurements that make up the metric and how these are calculated, a discussion of the key principles related to the metric, a discussion of the example graphic, and some "rules of thumb" for use during metrics analysis. Further information on general metrics analysis topics can be obtained from the PSM Part 4, Applying Software Measures [PSM]. In each of the subsequent charts, the acronym CRP stands for Current Reporting Period. Calculations for each of the example metrics described in this section are provided in Appendix A. The numbers circled in black in Table 3-1 are one-to-one traceable to the justification for the metric described in Section 2.2, *Selection and Justification of OO unique Metrics.*

A Note on Thresholds

> *Many of the metrics diagrams provided in the following sections contain thresholds. These thresholds have been provided as guidance only and are based on either (1) published best practices or (2) the experiences of the author. All thresholds are to be applied at the discretion of the reader, no threshold value is absolute, and all threshold values are subject to tailoring. It is the author's opinion, however, that where thresholds are supplied in the metrics diagrams, they should be incorporated as part of the metrics program, as tailored by the using organization. The application of thresholds is to ensure that some action is taken when a threshold is exceeded. When thresholds do not exist, there is no indication of what would be considered "good" or "bad," and, therefore, no indication of when action needs to be considered.*

Table 3-1. Object-Oriented Metrics

ISSUE	CATEGORY	RECOMMENDED METRICS
Growth and Stability	1 Size	Plan vs Actual Use Cases Completed Plan vs Actual Classes Completed Number of Attributes in a Class Number of Methods in a Class Number Scenarios in a Use Case
	2 Requirements Volatility	Added, Deleted and Modified Use Cases
	3 Design Volatility	Added, Deleted and Modified Classes Added, Deleted and Modified Methods Added, Deleted and Modified Attributes
Product Quality	4 Inheritance	Number of Children per Class Depth of Inheritance Tree per Class
	5 Object Structure	Weighted Methods per Class Type of Methods in Class
	6 Coupling	Coupling Between Classes Response for a Class
Schedule and Progress	7 Milestone	Plan vs Actual Milestone Days Milestone Slip Ratio
	8 Class Status	Plan vs Actual Classes Completed Plan vs Actual Methods Completed Plan vs Actual Attributes Completed Class Traceability Status Integration Test Traceability Status Plan vs Actual Classes that have Successfully Passed Integration Test
	9 Use Case Status	Plan vs Actual Use Cases Completed Use Case Traceability Status Functional Test Traceability Status Plan vs Actual Use Cases that have Successfully Passed Functional Test
	10 Build Content - Classes	Plan vs Actual Classes in Build Ratio of Classes in Build
	11 Build Content - Use Cases	Plan vs Actual Use Cases in Build Ratio of Use Cases in Build

3.1 Growth and Stability

The numbers circled in black in Table 3.1-1 are one-to-one traceable to the justification for the metric described in Section 2.2, *Selection and Justification of OO unique Metrics*.

Table 3.1-1 identifies each of the metrics selected for Growth and Stability.

Table 3.1-1 Growth and Stability OO Metrics

ISSUE	CATEGORY	RECOMMENDED METRICS
Growth and Stability	1 Size	Plan vs Actual Use Cases Completed Plan vs Actual Classes Completed Number of Attributes in a Class Number of Methods in a Class Number Scenarios in a Use Case
	2 Requirements Volatility	Added, Deleted and Modified Use Cases
	3 Design Volatility	Added, Deleted and Modified Classes Added, Deleted and Modified Methods Added, Deleted and Modified Attributes

3.1.1 Size

Size measures the physical size of a software product. Product size is a critical factor for estimating development schedule and cost. Size measures also provide information about the amount and frequency of change to a software product, which is especially critical late in the development.

3.1.1.1 Plan versus Actual Use Cases Completed

Issue	Growth and Stability
Category	Size
Measure	Planned number of use cases to be completed during the CRP Actual number of use cases completed during the CRP Calculate cumulative number of use cases planned to be complete (ref. App. A, (a)) Calculate cumulative number of use cases actually completed (ref. App. A, (a))
Description	Use cases describe the functionality required of the system based on the user viewpoint. The number and magnitude of the use cases defined for a particular application then becomes an indicator of software size. Unplanned additions and changes to the number and magnitude of use cases can adversely influence schedules and costs.
Example Graph	A line chart combined with a bar chart (Figure 3.1-1) is used to present the size information. The information presented includes the planned number of use cases and the actual number of use cases plotted over time. A cumulative value is included in the line chart and a CRP plan/actual is provided in the bar charts.
Performance Analysis	Actual values should track to the plan. Deviations of actuals above the plan line indicate that more is being accomplished than was originally projected. Deviations of actuals below the plan line indicate that less is being accomplished than originally projected.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	3	5	7	4	6	10	6	4	7	3	1	1
Actual	1	3	6	3	7	8	9	5	7	3	2	1
Cum Plan	3	8	15	19	25	35	41	45	52	55	56	57
Cum Act	1	4	10	13	20	28	37	42	49	52	54	55

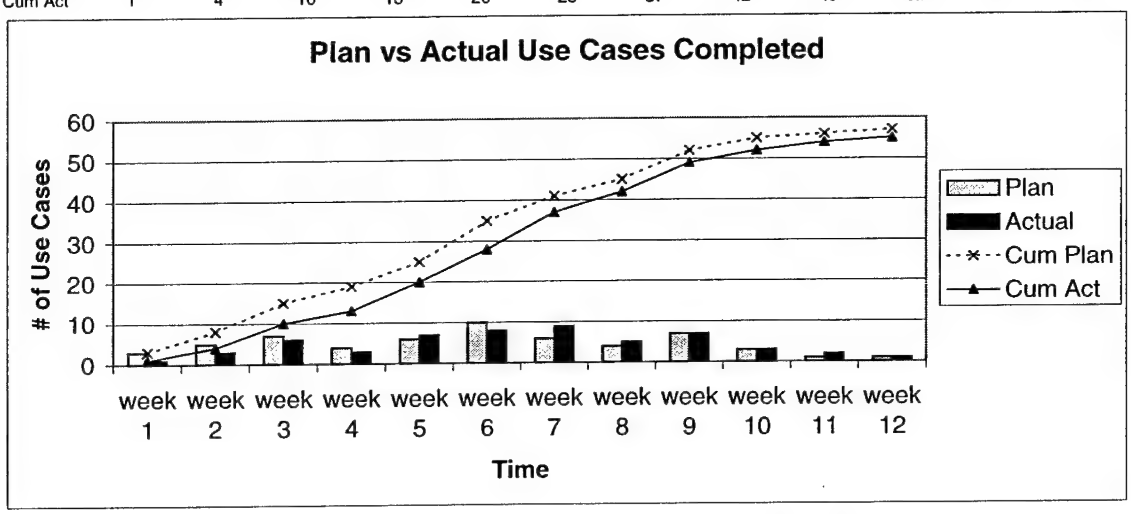


Figure 3.1-1. Plan versus actual use cases completed.

3.1.1.2 Plan versus Actual Classes Completed

Issue	Growth and Stability
Category	Size
Measure	Planned number of classes to be completed during the CRP Actual number of classes completed during the CRP Calculate cumulative number of classes planned to be complete (ref. App. A, (b)) Calculate cumulative number of classes actually completed (ref. App. A, (b))
Description	This indicator provides an estimate of software size in terms of design components. Unplanned additions and changes to the number and magnitude of classes can adversely influence schedules and costs.
Example Graph	A line chart combined with a bar chart (Figure 3.1-2) is used to present the size information. The information presented includes the planned number of classes and the actual number of classes plotted over time. A cumulative value is included in the line chart and a CRP plan/actual is provided in the bar charts.
Performance Analysis	Actual values should track to the plan. Deviations of actuals above the plan line indicate that more is being accomplished than was originally projected. Deviations of actuals below the plan line indicate that less is being accomplished than originally projected.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	5	7	10	6	8	12	8	6	9	5	2	1
Actual	4	5	8	5	9	12	11	6	9	4	3	2
Cum Plan	5	12	22	28	36	48	56	62	71	76	78	79
Cum Act	4	9	17	22	31	43	54	60	69	73	76	78

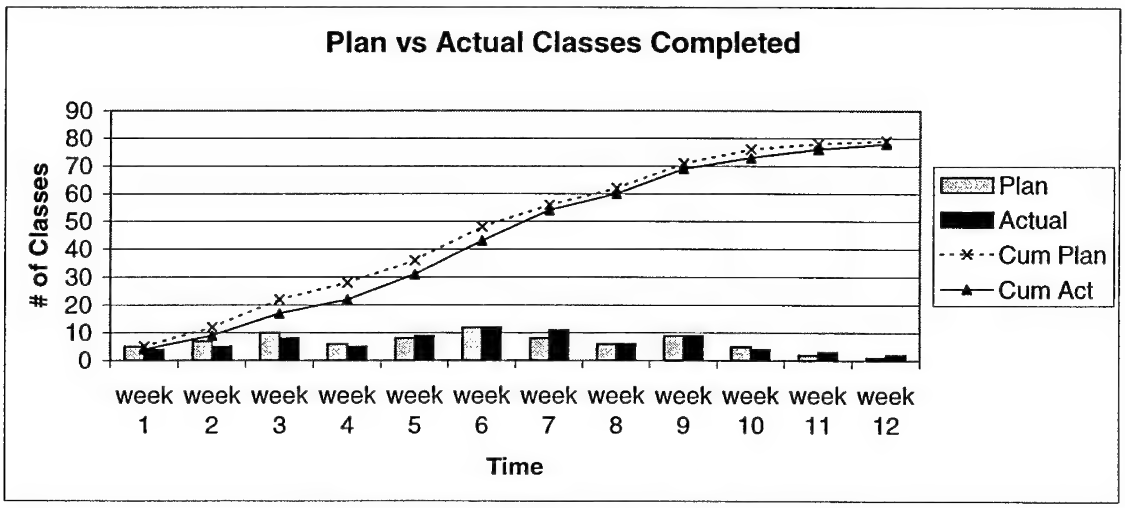


Figure 3.1-2. Plan versus actual classes completed.

3.1.1.3 Number of Attributes in a Class

Issue	Growth and Stability
Category	Size
Measure	CRP number of attributes in each class CRP number of classes Calculate cumulative number of classes/ cumulative number of attributes (ref. App. A, (c)) Calculate CRP number of classes/ CRP number of attributes (ref. App. A, (c))
Description	The number of attributes in a class is one measure of its size. A class that has multiple data objects may indicate that the class has a number of unnecessary specialized relationships with other classes in the system. Lower threshold calculated value should not be below 50%.
Example Graph	A line chart combined with a bar chart (Figure 3.1-3) is used to present the size information. The information presented includes the cumulative ratio value of attributes in a class. This is augmented by a bar chart indication of the CRP ratio value. Lower thresholds are indicated on the chart as well.
Performance Analysis	A large number of attributes in a class are an indication that the class may be doing too much. For values falling below the threshold, the unique class may be unnecessary. Review the system classes for optimization through combination.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# Attributes	7	8	12	5	11	16	13	9	3	5	6	2
# Classes	4	5	8	5	9	12	11	6	9	4	3	2
Cum Attributes	7	15	27	32	43	59	72	81	84	89	95	97
Cum Classes	4	9	17	22	31	43	54	60	69	73	76	78
Ratio	0.57	0.63	0.67	1.00	0.82	0.75	0.85	0.67	3.00	0.80	0.50	1.00
Cum Ratio	0.57	0.60	0.63	0.69	0.72	0.73	0.75	0.74	0.82	0.82	0.80	0.80
Upper Bound	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75	0.75
Lower Bound	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

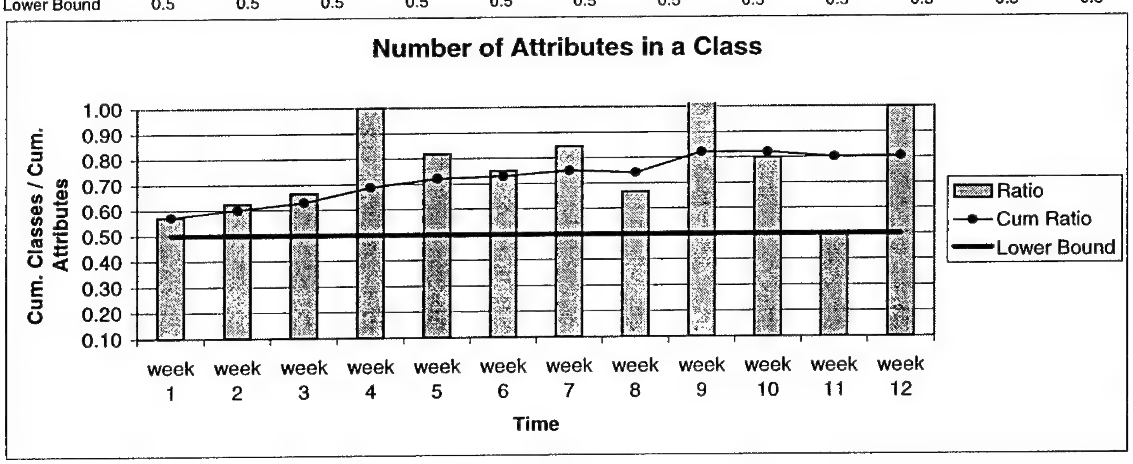


Figure 3.1-3. Number of attributes per class.

3.1.1.4 Number of Methods in a Class

Issue	Growth and Stability
Category	Size
Measure	CRP number of methods in each class CRP number of classes Calculate cumulative number of classes/ cumulative number of methods (ref. App. A, (d)) Calculate CRP Number of classes/ CRP Number of methods (ref. App. A, (d))
Description	The number of methods in a class is one measure of its size. The number of class methods indicates the amount of commonality being handled. A high number can indicate poor design when global services are used for all functions. Upper threshold calculated value should not exceed 80%.
Example Graph	A line chart combined with a bar chart (Figure 3.1-4) is used to present the size information. The information presented includes cumulative ratio value of methods in a class. This is augmented by a bar chart indication of the CRP ratio value. The upper threshold is indicated on the chart as well.
Performance Analysis	This metric measures the degree of specialization for a generic object. For values above the threshold, review the class methods looking for behavior that should be specific to the class. There is no minimum value threshold for methods in aclass.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# Methods	9	10	15	7	15	17	15	12	5	7	8	3
# Classes	4	5	8	5	9	12	11	6	9	4	3	2
Cum Methods	4	9	17	22	31	43	54	60	69	73	76	78
Cum Classes	9	19	34	41	56	73	88	100	105	112	120	123
Ratio	0.44	0.50	0.53	0.71	0.60	0.71	0.73	0.50	1.80	0.57	0.38	0.67
Cum Ratio	0.44	0.47	0.50	0.54	0.55	0.59	0.61	0.60	0.66	0.65	0.63	0.63
Upper Bound	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8	0.8

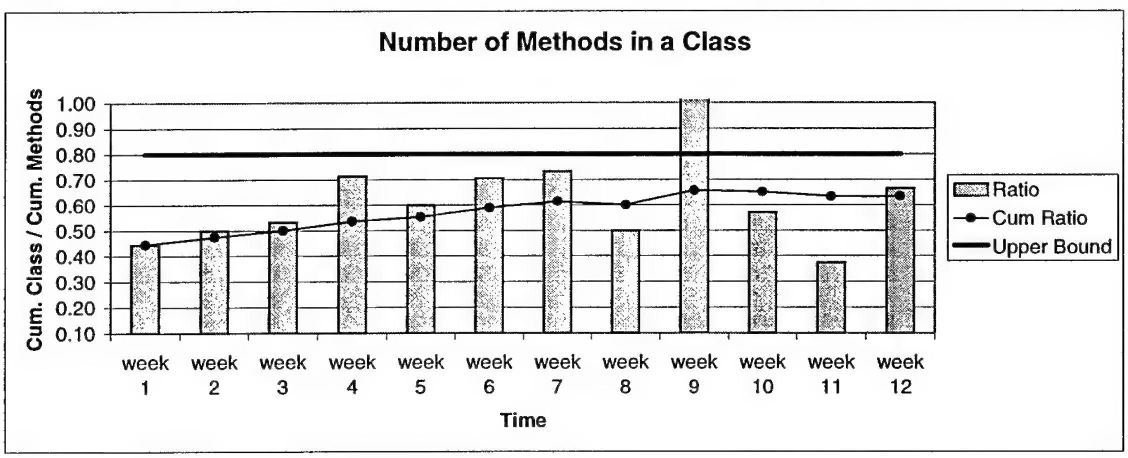


Figure 3.1-4. Number of methods per class.

3.1.1.5 Number of Scenarios in a Use Case

Issue	Growth and Stability
Category	Size
Measure	CRP number of use cases CRP number of scenarios per use case Calculate CRP number of uses cases/ CRP number of scenarios (ref. App. A, (e)) Calculate cumulative number of use cases/ cumulative number of scenarios (ref. App. A, (e))
Description	The number of scenario scripts is an indication of the size of the application to be developed. The number of scenario scripts also relates to the number of test cases that must be written to fully exercise the system. Upper value threshold should not exceed 60%. Lower value threshold should not exceed 40%.
Example Graph	A line chart combined with a bar chart (Figure 3.1-5) is used to present the size information. The information presented includes cumulative ratio value of scenarios in use case. This is augmented by a bar chart indication of the CRP ratio value. Thresholds are indicated on the chart as well.
Performance Analysis	The number of scenarios in a use case is an indication of user functionality provided. A scenario-to-use-case ratio that falls above the 60% threshold is an indication that too much functionality is being described in a single use case. An attempt should be made to simplify the design. Should this value fall below the indicated 40% threshold, the use cases should be reviewed for optimization opportunities.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# Scenarios	3	8	9	6	8	8	8	8	3	5	2	2
# Use Cases	1	3	4	4	5	5	2	3	5	1	1	1
Cum Scenarios	1	4	8	12	17	22	24	27	32	33	34	35
Cum Use Cases	3	11	20	26	34	42	50	58	61	66	68	70
Ratio	0.33	0.38	0.44	0.67	0.63	0.63	0.25	0.38	1.67	0.20	0.50	0.50
Cum Ratio	0.33	0.36	0.40	0.46	0.50	0.52	0.48	0.47	0.52	0.50	0.50	0.50
Upper Bound	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
Lower Bound	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4	0.4

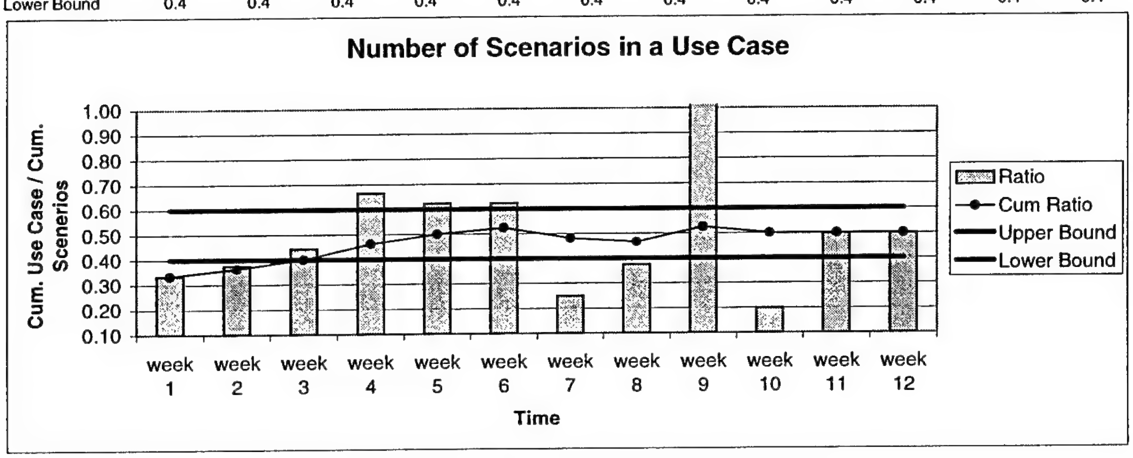


Figure 3.1-5. Number of scenarios per use case.

3.1.2 Requirements Volatility

Use Case volatility measures the stability of the required functionality of a software product. Use case volatility may be used to estimate development schedule and cost. This measure provides information about the amount and frequency of change to software functionality, which is especially critical late in the development. Use case volatility generally correlates with effort, cost, schedule, and product size changes.

Many of the volatility metrics rely on a "base" size value being included in the calculations. The "base" value is used as the basis on which the requirements design or code will be modified. This concept is necessary for use with incremental, evolutionary, or reuse-driven developments as the requirements, design, and code are not being written from scratch but rely on existing products for modification. The "base" value is that portion of the integrated software product that would be considered pre-existing. During the initial iteration of the product life cycle for newly developed software, this value would be set to zero.

The churn ratios calculated on the subsequent metrics charts reflect a normalized numeric value associated with how much of the product has been modified over time. A churn ratio approaching 1 reflects that the product has been designed more than once. This can be misleading due to the fact that you can have components of the product that are designed many times, whereas other portions of the product may have only been designed once.

3.1.2.1 Added, Deleted and Modified Use Cases

Issue	Growth and Stability
Category	Requirements Volatility
Measure	CRP planned number of use cases CRP actual base number of use cases CRP actual added number of use cases CRP actual deleted number of use cases CRP modified number of use cases (which includes all changed aspects of the use case definition) Calculate cumulative planned number of use cases (ref. App. A, (f)) Calculate cumulative actual number of use cases (ref. App. A, (f)) Calculate churn ratio (scaled ratio of CRP modified use cases to cummulative actual use cases) (ref. App. A, (f))
Description	The Added, Deleted Modified Use-Case indicator can be used to monitor changes to requirements throughout a project, which can serve as a leading indictor of delays, cost increases and rework. The churn ratio is provided as an indicator of the amount of rework being accomplished for a given use case.
Example Graph	A line chart combined with a bar chart (Figure 3.1-6) is used to present the volatility information. The information presented includes the cumulative number of use cases. This is augmented by a bar chart indication of the CRP number of modified, added and deleted use cases.
Performance Analysis	A high level of use case volatility may require adjustment to current resource allocations, effort estimates, budgets, and schedule. The churn ratio should be consistent with the system phase. (Sometimes it is a measure of rework, other times expected work.) In addition, adding and deleting of use cases late in a project could indicate an unstable analysis process.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	3	5	7	4	6	10	6	4	7	3	1	1
Base	0	0	0	0	0	0	0	0	0	0	0	0
Added	1	7	10	5	8	8	9	12	10	4	6	2
Deleted	0	3	2	7	0	2	1	4	0	4	1	1
Modified	2	3	8	5	4	12	15	30	8	9	26	10
Actual	1	4	8	-2	8	6	8	8	10	0	5	1
Cum Plan	3	8	15	19	25	35	41	45	52	55	56	57
Cum Actual	1	5	13	11	19	25	33	41	51	51	56	57
Churn Ratio	2.00	0.60	0.62	0.45	0.21	0.48	0.45	0.73	0.16	0.18	0.46	0.18

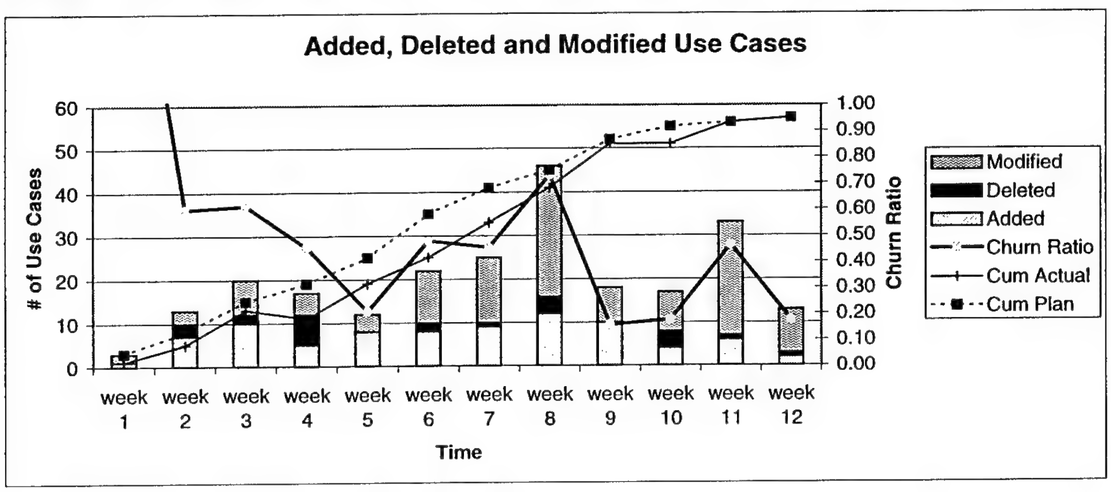


Figure 3.1-6. Added, deleted and modified use cases.

3.1.3 Design Volatility

Class volatility measures the stability of the design of a software product. Class volatility can be used to estimate downstream testing and rework costs. Method/attribute volatility measures the stability of a software product. These measures provide information about the amount and frequency of change to software design, which is especially critical late in the development. Method/attribute and class volatility generally correlates to effort, cost, schedule, and product size changes.

See paragraph 3.1.2 for a description of the concepts of "base" and "churn ratio."

3.1.3.1 Added, Deleted and Modified Classes

Issue	Growth and Stability
Category	Design Volatility
Measure	CRP planned number of classes CRP actual base number of classes CRP actual added number of classes CRP actual deleted number of classes CRP modified number of classes (which includes all changed aspects of the class definition) Calculate cumulative planned number of classes (ref. App. A, (g)) Calculate cumulative actual number of classes (ref. App. A, (g)) Calculate churn ratio (scaled ratio of modified classes to base) (ref. App. A, (g))
Description	This indicator can be used to monitor changes to design throughout a project, which can serve as a leading indictor of delays, cost increases and rework. The churn ratio is provided as an indicator of the amount of rework being accomplished.
Example Graph	A line chart combined with a bar chart (Figure 3.1-7) is used to present the volatility information. The information presented includes the cumulative number of classes. This is augmented by a bar chart indication of the number of CRP modified, added and deleted classes.
Performance Analysis	A high level of class volatility may require adjustment to current resource allocations, effort estimates, budgets, and schedule. The churn ratio should be consistent with the system phase. (Sometimes it is a measure of rework, other times expected work.) In addition, adding and deleting of classes late in a project could indicate an unstable design process.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	3	5	7	4	6	10	6	4	7	3	1	1
Base	0	0	0	0	0	0	0	0	0	0	0	0
Added	1	6	10	9	12	6	6	6	8	12	5	3
Deleted	0	3	2	7	0	2	1	4	0	4	1	1
Modified	2	2	6	7	6	6	10	7	12	20	30	20
Actual	1	3	8	2	12	4	5	2	8	8	4	2
Cum Plan	3	8	15	19	25	35	41	45	52	55	56	57
Cum Actual	1	4	12	14	26	30	35	37	45	53	57	59
Churn Ratio	2.00	0.50	0.50	0.50	0.23	0.20	0.29	0.19	0.27	0.38	0.53	0.34

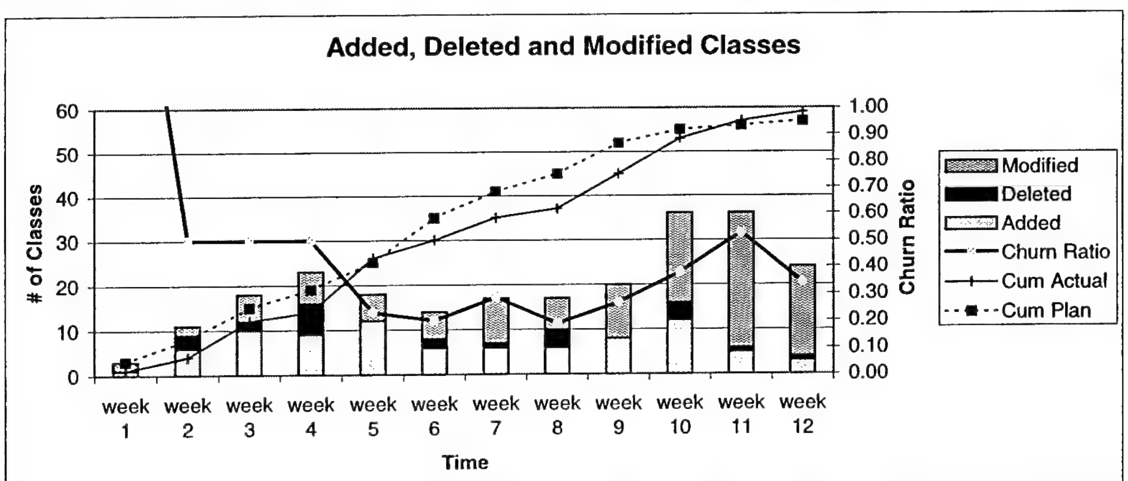


Figure 3.1-7. Added, deleted and modified classes.

3.1.3.2 Added, Deleted and Modified Methods

Issue	Growth and Stability
Category	Design Volatility
Measure	CRP planned number of methods CRP actual base number of methods CRP actual added number of methods CRP actual deleted number of methods CRP actual modified number of methods (which includes all changed aspects of the method description) Calculate cumulative planned number of methods (ref. App. A, (i)) Calculate cumulative actual number of methods (ref. App. A, (i)) Calculate churn ratio (scaled ratio of modified methods to base) (ref. App. A, (i))
Description	This indicator can be used to monitor changes to design throughout a project, which can serve as a leading indictor of delays, cost increases and rework. The churn ratio is provided as an indicator of the amount of rework being accomplished for a given method.
Example Graph	A line chart combined with a bar chart (Figure 3.1-8) is used to present the volatility information. The information presented includes the cumulative number of methods. This is augmented by a bar chart indication of the CRP numbers of modified, added and deleted methods.
Performance Analysis	A high level of method volatility may require adjustment to current resource allocations, effort estimates, budgets, and schedule. The churn ratio should be consistent with the system phase. (Sometimes it is a measure of rework, other times expected work.) In addition, adding and deleting of methods late in a project could indicate an unstable design process.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	5	7	10	6	8	12	8	6	9	5	2	1
Base	0	0	0	0	0	0	0	0	0	0	0	0
Added	5	6	12	14	12	10	8	8	8	14	8	2
Deleted	1	1	2	7	0	2	3	5	0	5	1	1
Modified	2	2	3	3	5	4	6	12	15	5	1	0
Actual	4	5	10	7	12	8	5	3	8	9	7	1
Cum Plan	5	12	22	28	36	48	56	62	71	76	78	79
Cum Actual	4	9	19	26	38	46	51	54	62	71	78	79
Churn Ratio	0.50	0.22	0.16	0.12	0.13	0.09	0.12	0.22	0.24	0.07	0.01	0.00

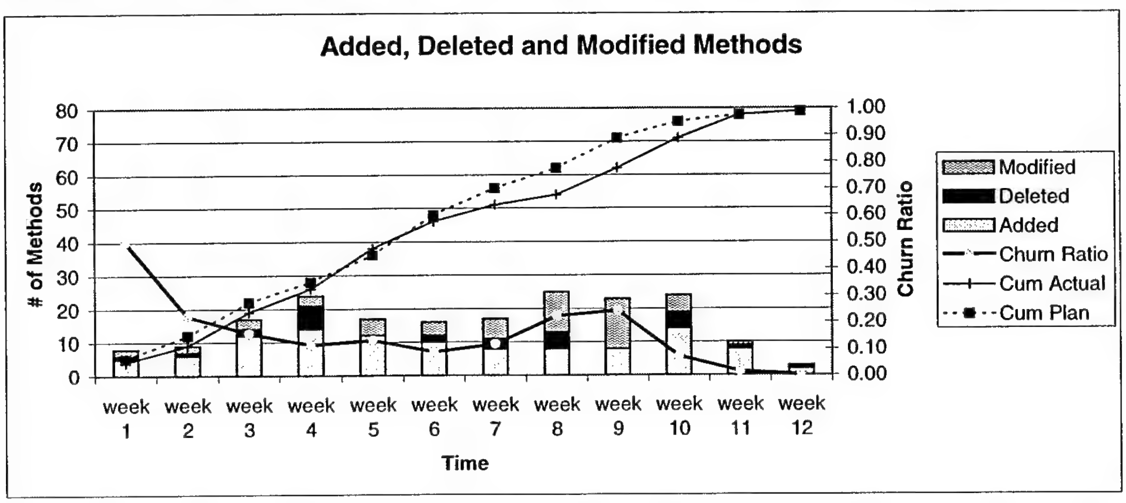


Figure 3.1-8. Added, deleted and modified methods.

3.1.3.3 Added, Deleted and Modified Attributes

Issue	Growth and Stability
Category	Design Volatility
Measure	CRP planned number of attributes CRP actual base number of attributes CRP actual added number of attributes CRP actual deleted number of attributes CRP actual modified number of attributes (which includes all changed aspects of the attribute definition) Calculate cumulative planned number of attributes (ref. App. A, (k)) Calculate cumulative actual number of attributes (ref. App. A, (k)) Calculate churn ratio (scaled ratio of modified methods to base) (ref. App. A, (k))
Description	This indicator can be used to monitor changes to design throughout a project, which can serve as a leading indictor of delays, cost increases and rework. The churn ratio is provided as an indicator of the amount of rework being accomplished for a given attribute.
Example Graph	A line chart combined with a bar chart (Figure 3.1-9) is used to present the volatility information. The information presented includes the cumulative number of attributes. This is augmented by a bar chart indication of the CRP number of modified, added and deleted attributes.
Performance Analysis	A high level of attribute volatility may require adjustment to current resource allocations, effort estimates, budgets, and schedule. The churn ratio should be consistent with the system phase. (Sometimes it is a measure of rework, other times expected work.) In addition, adding and deleting of attributes late in a project could indicate an unstable design process.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	5	7	10	6	8	12	8	6	9	5	2	1
Base	0	0	0	0	0	0	0	0	0	0	0	0
Added	5	6	8	10	10	12	10	15	13	9	6	5
Deleted	1	1	2	7	0	2	3	5	0	5	1	1
Modified	2	2	3	10	6	8	12	16	19	19	7	9
Actual	4	5	6	3	10	10	7	10	13	4	5	4
Cum Plan	5	12	22	28	36	48	56	62	71	76	78	79
Cum Actual	4	9	15	18	28	38	45	55	68	72	77	81
Churn Ratio	0.50	0.22	0.20	0.56	0.21	0.21	0.27	0.29	0.28	0.26	0.09	0.11

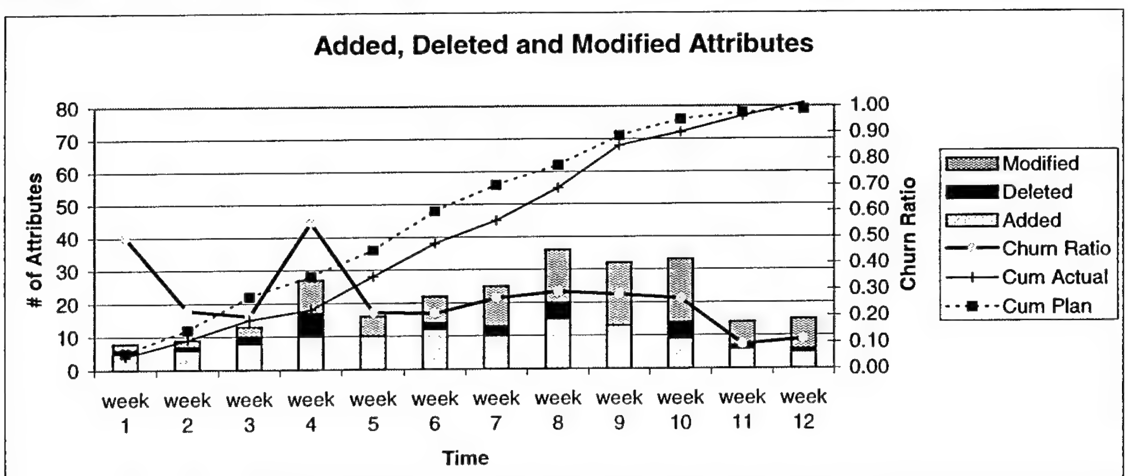


Figure 3.1-9. Added, deleted and modified attributes.

3.2 Product Quality

The numbers circled in black in Table 3.2-1 are one-to-one traceable to the justification for the metric described in Section 2.2 *Selection and Justification of OO unique Metrics.* Table 3.2-1 identifies the metrics selected for Product Quality.

Table 3.2-1. Product Quality Metrics

ISSUE	CATEGORY	RECOMMENDED METRICS
Product Quality	4 Inheritance	Number of Children per Class Depth of Inheritance Tree per Class
	5 Object Structure	Weighted Methods per Class Type of Methods in Class
	6 Coupling	Coupling Between Classes Response for a Class

3.2.1 Inheritance

Inheritance measures the structure of software components, based on the number of children. Complex components are generally harder to test, are more difficult to maintain, and may contain more defects than less complex components. Inheritance measures may provide indications of the need to redesign. In addition, inheritance metrics can be used to ascertain the effects of change to a given component. A seemingly small change to a parent can affect all of the children and grandchildren adversely. Determining the inheritance for a class can help establish how widespread changes to a parent class actually are.

3.2.1.1 Number of Children per Class

Issue	Product Quality
Category	Inheritance
Measure	Count the number of children for each class Sum the number of classes having the same number of children
Description	The number of children per class measures the number of immediate subclasses subordinated to a class in the class hierarchy. This is a measure of the horizontal breadth of the class structure. Threshold: 80% of the classes in the system should have 7 or more inherited subclasses.
Example Graph	A histogram (Figure 3.2-1) is used to present the structure information. The information presented includes the number of classes containing a particular number of children classes.
Performance Analysis	The percentage of children inherited should be high. The greater the number of children, the greater the reuse, since inheritance is a form of reuse. There is no lower threshold for this metric.

	9	8	7	6	5	4	3	2	1	0	Median	Mode	Average
Count	150	100	80	50	30	20	0	0	0	0	8	9	7.46

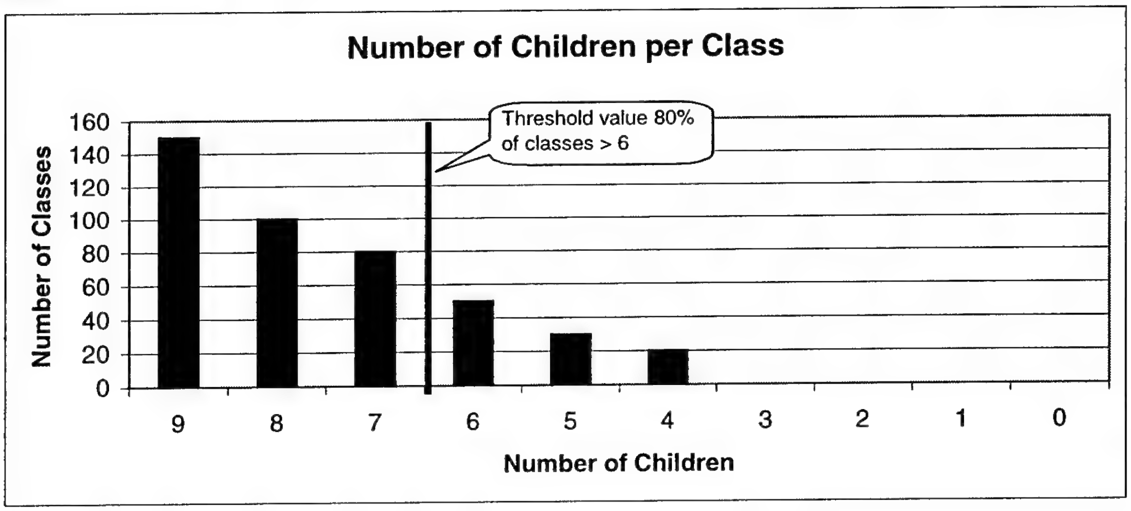


Figure 3.2-1. Number of children per class.

3.2.1.2 Depth of Inheritance Tree (DIT) per Class

Issue	Product Quality
Category	Inheritance
Measure	Count the number of nodes from each class to the root of the inheritance tree Sum the number of classes containing the same number of nodes between it and the root of the inheritance tree
Description	In cases involving multiple inheritance, the DIT will be the maximum length from the node to the root of the tree. This is a measure of the vertical depth of the class structure. Threshold: 80% of the classes in the system should have less than 6 inheritance levels.
Example Graph	A histogram (Figure 3.2-2) is used to present the structure information. The information presented includes the number of classes with a particular class tree depth.
Performance Analysis	Large numbers of nesting levels in a class structure is an indication that too many classes have been created in the system. The deeper a class is in the hierarchy, the greater the number of methods it is likely to inherit, making it more difficult to predict its behavior. A nesting level of less than 6 should be maintained throughout the system.

	0	1	2	3	4	5	6	7	8	9	Median	Mode	Average
Count	100	75	75	50	40	30	20	10	0	0	2	0	2.19

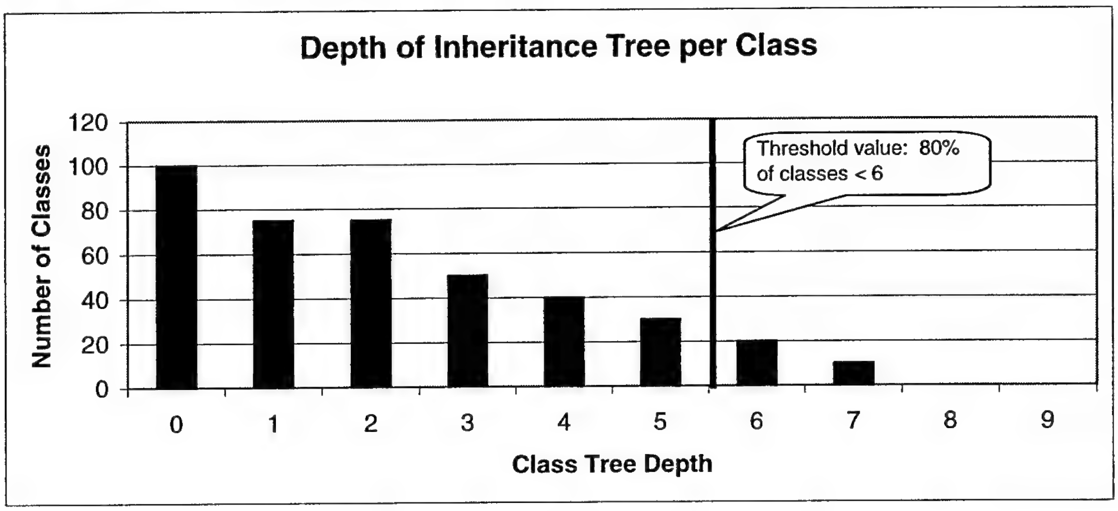


Figure 3.2-2. Depth of inheritance tree per class.

3.2.2 Object Structure

The structure of software is measured using object structure measures. Complex objects are generally harder to test, are more difficult to maintain, and may contain more defects than less complex objects. Object structure measures provide indications of the need to redesign and of the relative amount of testing required.

3.2.2.1 Weighted Methods per Class

Issue	Product Quality
Category	Object Structure
Measure	Identify the complexity of each method (by using a static code analyzer tool) [MCCABE] For each class sum the complexity for each method in the class into a class complexity Sum the number of classes containing equivalent levels of class complexity
Description	This measure describes the complexity of a class through the complexity of its methods. Threshold: class complexity should not exceed 70.
Example Graph	A histogram (Figure 3.2-3) is used to present the structure information. The information presented includes the number of classes having a particular class complexity.
Performance Analysis	A more complex class is more difficult to maintain. There are reasons, however, to have classes that have a higher degree of complexity. Overall, however, the majority of the system should not be made up of highly complex classes. Classes that exhibit a cumulative complexity of 70 or over should be examined to ensure that their complexity is justified. When this occurs it is an indication that all the classes should be reviewed to ensure that only those classes required to be highly complex are.

	0-9	10-19	20-29	30-39	40-49	50-59	60-69	70-79	80-89	>90	Median	Mode	Average
Count	100	75	75	50	40	30	20	10	0	0	20	0	21.88

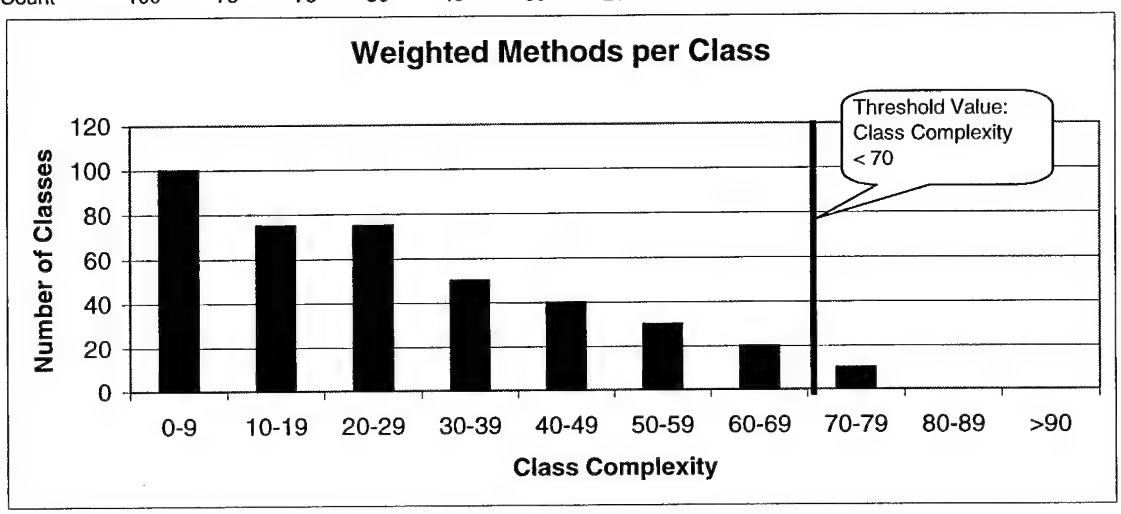


Figure 3.2-3. Weighted methods per class.

3.2.2.2 Type of Methods in Class

Issue	Product Quality
Category	Object Structure
Measure	Number of CRP classes Number of CRP methods per class Number of CRP private methods per class Number of CRP protected methods per class Number of CRP public methods per class Calculate cumulative number or private methods per class (ref. App. A, (l)) Calculate cumulative number of protected methods per class (ref. App. A, (l)) Calculate cumulative number of public methods per class (ref. App. A, (l))
Description	The number of public methods in a class is a measure of the amount of system functionality being provided by the class. In addition, the number of public methods is a reflection of the total number of methods provided by the class, because each public method is supported by some number of private methods. Threshold value: Non-public methods should not exceed 80% of design.
Example Graph	A 100% stacked column (Figure 3.2-4) is used to present the method type information. The information presented compares the percentage of each method type that contributes to the total across the method types.
Performance Analysis	The comparison of public to non-public methods in a class is an indication of the amount of work being performed by the class. As a goal at least 20% of all methods should be public. In addition, classes with large percentages of public methods should be examined to determine if some of those should be made private or protected.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# Classes	4	5	8	5	9	12	11	6	9	4	3	2
# Methods	9	10	15	7	15	17	15	12	5	7	8	3
# Private	4	5	7	4	9	12	8	7	3	5	4	1
# Protected	3	3	5	1	3	4	6	3	1	0	2	1
# Public	2	2	3	2	3	1	1	2	1	2	2	1
Cum Private	4	9	16	20	29	41	49	56	59	64	68	69
Cum Protected	3	6	11	12	15	19	25	28	29	29	31	32
Cum Public	2	4	7	9	12	13	14	16	17	19	21	22

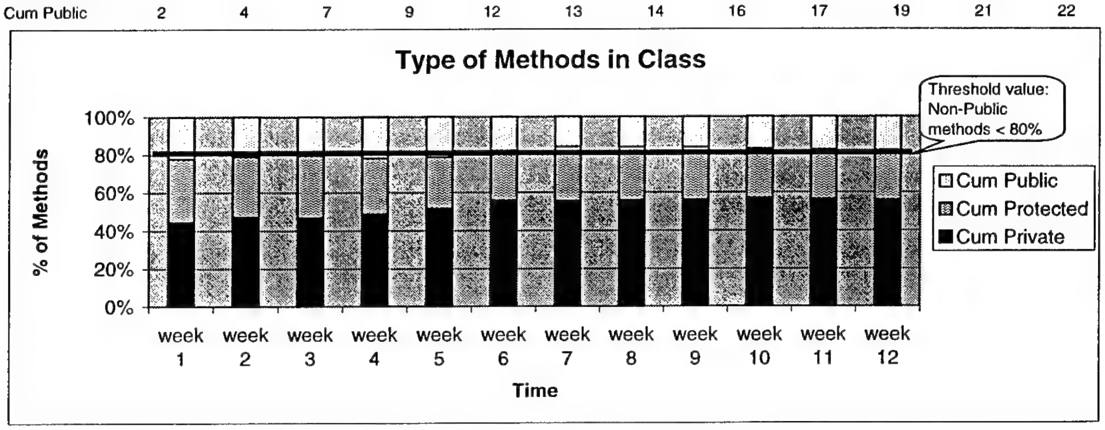


Figure 3.2-4. Type of methods class.

3.2.3 Coupling

Coupling measures the degree to which the classes of the system interchange information. There are two measures of coupling. The first is the amount of data that is passed between the classes. The second is the number of responses possible for any given method invocation. A system with a high degree of coupling can be difficult to maintain.

3.2.3.1 Coupling Between Classes

Issue	Product Quality
Category	Coupling
Measure	Count the number of methods called per class Sum the number of classes calling a particular number of methods
Description	This metric provides a profile of the number of other classes with which a particular class communicates (i.e.,., counts the number of external methods called by a particular class).
Example Graph	A histogram (Figure 3.2-5) is used to present the structure information. The information presented includes the number of classes that call a particular number of methods.
Performance Analysis	Excessive coupling between object classes is detrimental to modular design and prevents reuse. The more independent a class is, the easier it is to reuse in another application. In order to improve modularity and promote encapsulation, class coupling should be kept to a minimum. The larger the number of couples, the higher the sensitivity to changes in other parts of the design. A measure of coupling is useful to determine how complex the testing of various parts of the design is likely to be. The higher the class coupling, the more rigorous the testing needs to be. No threshold has been defined for this metric.

	0-4	5-9	10-14	15-19	20-24	25-29	30-34	35-39	40-44	>45	Median	Mode	Average
Count	140	115	80	50	35	30	20	15	10	5	3	0	10.65

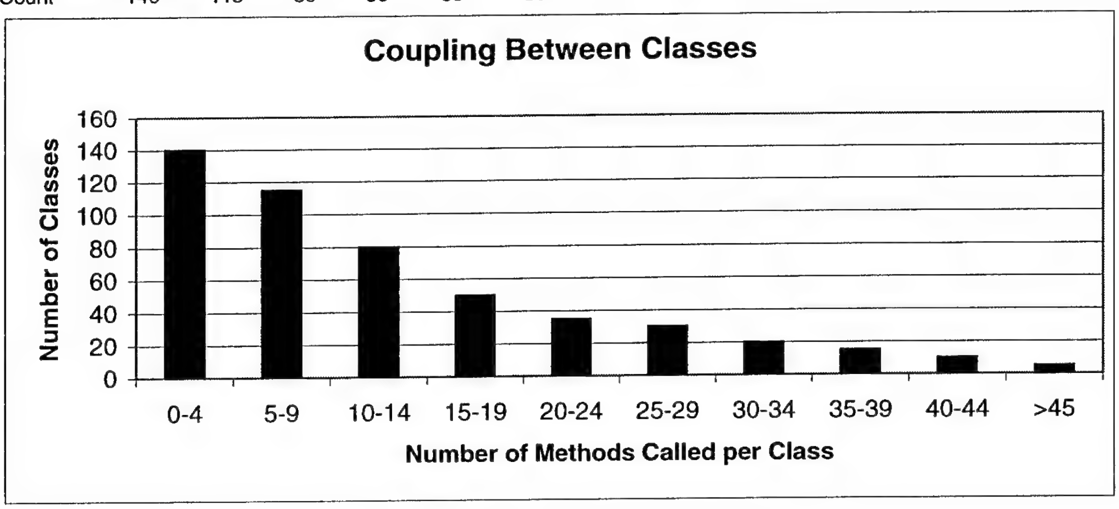


Figure 3.2-5. Coupling between classes.

3.2.3.2 Response for a Class(RFC)

Issue	Product Quality
Category	Coupling
Measure	Number of methods in other classes that can respond to any message sent by the class under consideration Number of classes that respond to the messages
Description	The response set of a class is a count of methods accessible to an object of this class type due to inheritance.
Example Graph	A histogram (Figure 3.2-6) is used to present the structure information. The information presented includes the number of classes having a particular number of methods in its response set.
Performance Analysis	This metric measures the degree of class coupling. A low level of coupling is necessary to reduce system complexity. Therefore, the optimal RFC metric would be to have the largest number of classes contained in the lower response categories. No thresholds have been provided for this metric.

	0	1	2	3	4	5	6	7	8	9	Median	Mode	Average
Count	50	100	75	65	50	30	20	10	0	0	2	0	2.19

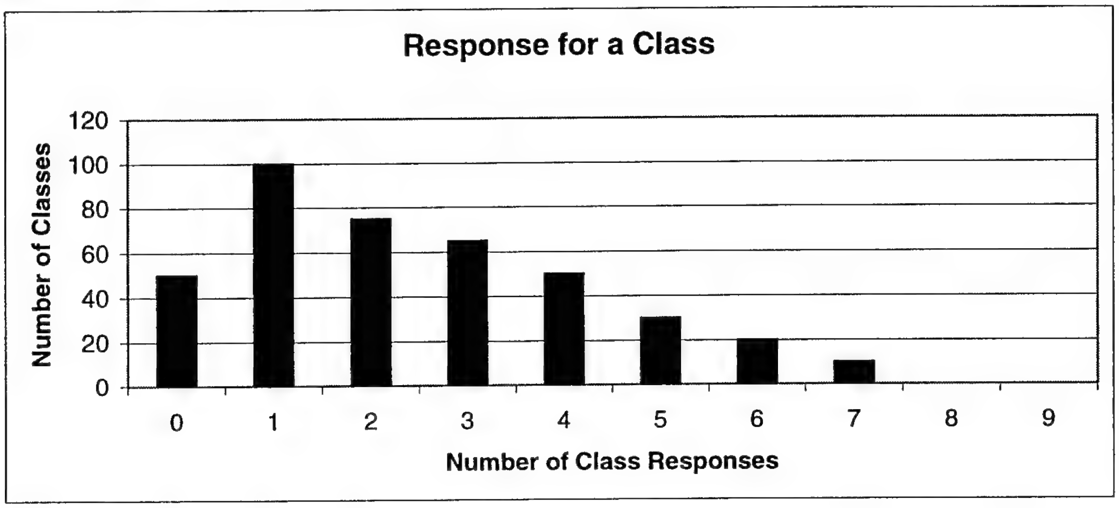


Figure 3.2-6. Response for a class.

3.3 Schedule and Progress

The numbers circled in black in Table 3.3-1 are one-to-one traceable to the justification for the metric described in Section 2.2, *Selection and Justification of OO unique Metrics*. Table 3.3-1 identifies the metrics selected for Schedule and Progress. The metrics included in the Schedule and Progress section must be linked to the Integrated Master Plan (IMP) and Integrated Master Schedule (IMS) to be useful in managing the OO development. [IMP/IMS] The prudent project manager of an OO development effort must ensure that the appropriate terminology is reflected in these program-level documents. In addition, the management system used to track the OO development, generically referred to as the Earned Value Management System (EVMS), must also reflect the terminology of the OO development.

Table 3.3-1. Schedule and Progress Metrics.

ISSUE	CATEGORY	RECOMMENDED METRICS
Schedule and Progress	7 Milestone	Plan vs Actual Milestone Days Milestone Slip Ratio
	8 Class Status	Plan vs Actual Classes Completed Plan vs Actual Methods Completed Plan vs Actual Attributes Completed Class Traceability Status Integration Test Traceability Status Plan vs Actual Classes that have Successfully Passed Integration Test
	9 Use Case Status	Plan vs Actual Use Cases Completed Use Case Traceability Status Functional Test Traceability Status Plan vs Actual Use Cases that have Successfully Passed Functional Test
	10 Build Content - Classes	Plan vs Actual Classes in Build Ratio of Classes in Build
	11 Build Content - Use Cases	Plan vs Actual Use Cases in Build Ratio of Use Cases in Build

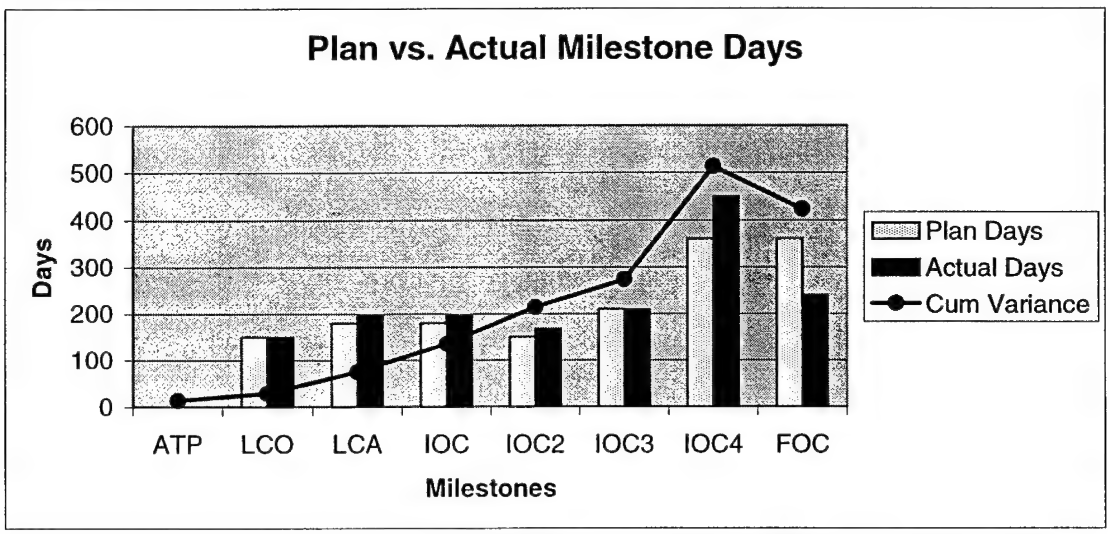
3.3.1 Milestone

The milestone performance measure provides basic schedule and progress information for key software development activities and events. The measures also help to identify and assess dependencies among software development activities and events. Monitoring changes in schedule allows the project manager to assess the risk in achieving future milestones. Milestone metrics are not unique to object-oriented development. However, when the OO-specific life-cycle model (described in Appendix C) is used, different milestones are defined with different names than the traditional milestones used in non-OO developments. (An example of an OO-specific life-cycle model would be that which is described in the Rational Unified Process (RUP). [RATIONAL]) The metrics described in this section have been adapted to use the OO-specific milestones. Some object-oriented implementations may, however, use the traditional milestone terminology, in which case the traditional milestone metrics would be used instead.

3.3.1.1 Plan vs. Actual Milestone Days

Issue	Schedule and Progress
Category	Milestone
Measure	Planned milestone completion date Actual milestone completion date Number of days between (planned versus actual) milestones (if this is the first milestone then this calculation is not applicable) Calculate total milestone variance and cumulative total milestone variance (ref. App. A, (m))
Description	This indicator helps identify the current status of major project events, and allows assessment of the impact of potential or actual schedule slips on future activities and milestones.
Example Graph	A line chart combined with a bar chart (Figure 3.3-1) is used to present the milestone slip information. The information presented by the line chart is a cumulative milestone slip variance. This is augmented by a bar chart showing the milestone Planned vs actual value.
Performance Analysis	Slips in activities and milestones on the critical path are of greatest concern, due to the ripple effect in later parts of the schedule. The graph should contain a sufficient level of detail to monitor progress. If multiple builds or releases are planned, separate activities and milestones should be defined for each build or release.

	ATP	LCO	LCA	IOC	IOC2	IOC3	IOC4	FOC
Plan Date	01/01/2000	05/31/2000	11/30/2000	05/31/2001	10/31/2001	05/31/2002	05/31/2003	05/31/2004
Plan Days		150	180	180	150	210	360	360
Actual Date	01/15/2000	06/15/2000	12/31/2000	07/15/2001	01/02/2002	07/31/2002	10/31/2003	06/30/2004
Actual Days		150	196	195	167	209	450	240
Late Start	14	15	30	45	62	60	150	30
Total Variance	14	15	46	60	79	59	240	-90
Cum Variance	14	29	75	135	214	273	513	423



ATP = Authority to Proceed, LCO = Life Cycle Objectives, LCA = Life Cycle Architecture, IOC = Initial Operational Capability, FOC = Final Operational Capability

Figure 3.3-1. Plan vs. actual milestone days.

3.3.1.2 Milestone Slip Ratio

Issue Category	Schedule and Progress Milestone
Measure	Planned milestone completion date. Actual milestone completion date Number of days between milestones (if this is the first milestone then this calculation is not applicable) Calculate milestone slip ratio, which is the ratio of the milestone variance to the planned days between milestones (ref. App. A, (n))
Description	This indicator helps identify the current status of the project. It provides visibility into the magnitude of the schedule slip when compared to the overall length of the schedule for each milestone
Example Graph	A line chart (Figure 3.3-2) is used to present the milestone slip information. The information presented includes a milestone slip value.
Performance Analysis	A ratio of zero is ideal, the larger the ratio, the worse the slip in proportion to the schedule.

	ATP	LCO	LCA	IOC	IOC[2]	IOC[3]	IOC[4]	FOC
Plan Date	01/01/2000	05/31/2000	11/30/2000	05/31/2001	10/31/2001	05/31/2002	05/31/2003	05/31/2004
Plan Days	10	150	180	180	150	210	360	360
Actual Date	01/15/2000	06/15/2000	12/31/2000	07/15/2001	01/02/2002	07/31/2002	10/31/2003	06/30/2004
Actual Days		150	196	195	167	209	450	240
Late Start	14	15	30	45	62	60	150	30
Variance	4	15	46	60	79	59	240	-90
Slip Ratio	0.40	0.10	0.26	0.33	0.53	0.28	0.67	-0.25

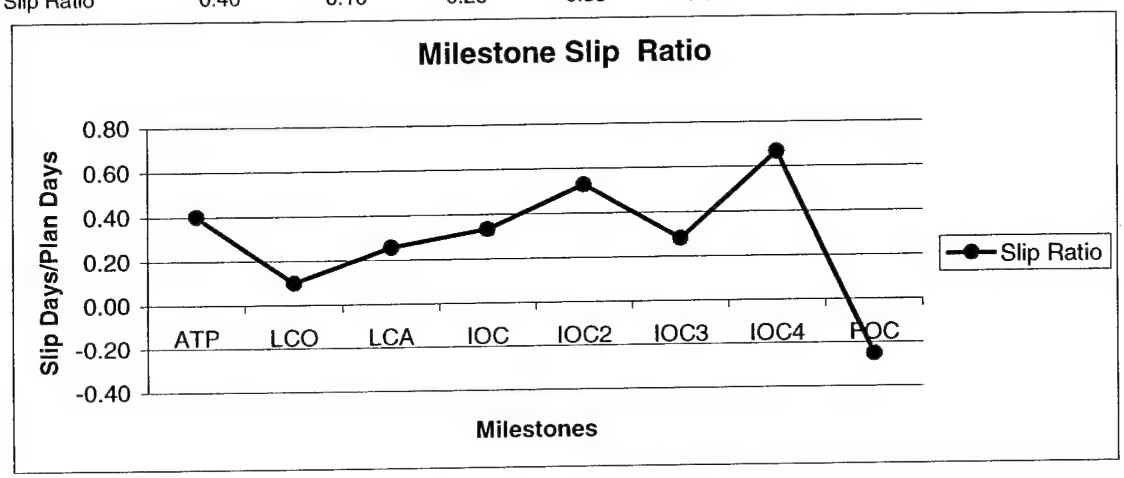


Figure 3.3-2. Milestone slip ratio.

3.3.2 Class Status

Class status measures address progress based on the completion of work units that combine incrementally to form a complete software activity or product. If objective completion criteria are defined, class status measures are extremely effective for assessing progress at any point in the project. Objective completion criteria are defined as "measurable and useful indicators that demonstrate that the achievement or maturity/progress in an activity or accomplishment has been achieved. Accomplishment criteria include, but are not limited to, (1) completed work efforts; (2) activities to confirm success of meeting technical, schedule, or cost parameters; (3) Internal documents that provide results of incremental verification, and (4) Completion of critical process activities and products." [IMP/IMS] These measures are used for projecting completion dates for the activity or product.

3.3.2.1 Plan versus Actual Classes Completed

Issue	Schedule and Progress
Category	Class Status
Measure	Refer to Section 3.1.1.2 (Issue: Growth and Stability, Attribute: Size).

3.3.2.2 Plan versus Actual Methods Completed

Issue	Schedule and Progress
Category	Class Status
Measure	Planned number of methods to be completed during the CRP Actual number of methods completed (i.e., coded and unit tested) during the CRP Calculate cumulative number of methods planned to be completed (ref. App. A, (h)) Calculate cumulative number of methods actually completed (ref. App. A, (h))
Description	This indicator provides an estimate of class status. Unplanned additions and changes to the number and magnitude of methods can adversely influence schedules and costs. This metric can be used to determine when the implementation phase is completed.
Example Graph	A line chart combined with a bar chart (Figure 3.3-3) is used to present the class status information. The information presented includes the planned number of methods and the actual number of methods plotted over time. A cumulative value is included in the line chart and a CRP plan/actual is provided in the bar charts.
Performance Analysis	The planned magnitude of the job must be realistic over the period of performance. Large changes in the rate per period should be evaluated for feasibility.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	5	7	10	6	8	12	8	6	9	5	2	1
Actual	4	5	8	5	9	12	11	6	9	4	3	2
Cum Plan	5	12	22	28	36	48	56	62	71	76	78	79
Cum Act	5	10	18	23	32	44	55	61	70	74	77	79

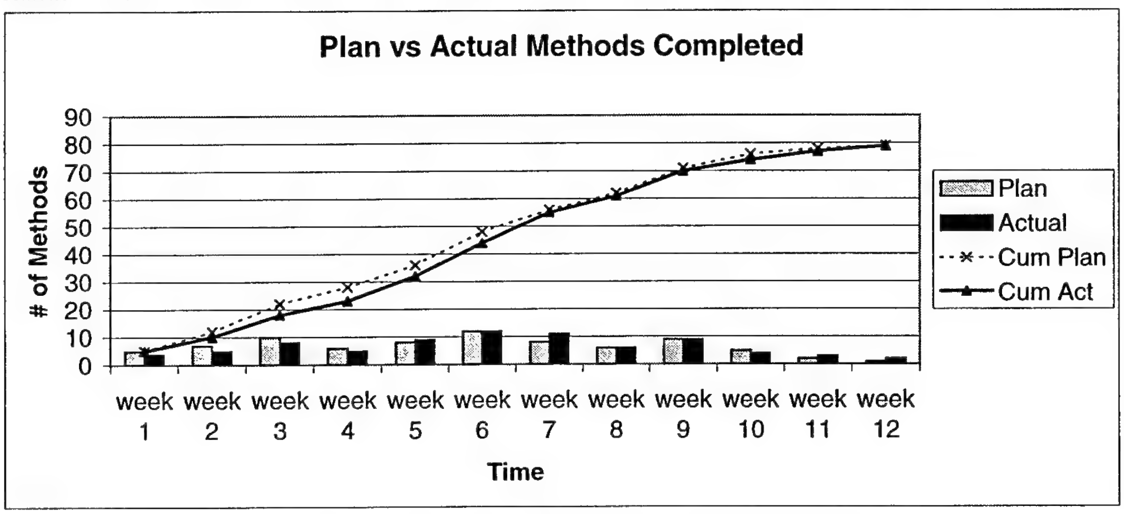


Figure 3.3-3. Plan versus actual methods completed.

3.3.2.3 Plan versus Actual Attributes Completed

Issue	Schedule and Progress
Category	Class Status
Measure	Planned number of attributes to be completed during the CRP Actual number of attributes completed (i.e., coded and unit tested) during the CRP Calculate cumulative number of attributes planned to be completed (ref. App. A, (j)) Calculate cumulative number of attributes actually completed (ref. App. A, (j))
Description	This indicator provides an estimate of the programming associated with the development of data structures within the class. Unplanned additions and changes to the number and magnitude of attributes can adversely influence schedules and costs.
Example Graph	A line chart combined with a bar chart (Figure 3.3-4) is used to present the class status information. The information presented includes the planned number of attributes and the actual number of attributes plotted over time. A cumulative value is included in the line chart and a CRP plan/actual is provided in the bar charts.
Performance Analysis	The planned magnitude of the job must be realistic over the period of performance. Large changes in the rate per period should be evaluated for feasibility.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	5	7	10	6	8	12	8	6	9	5	2	1
Actual	4	5	8	5	9	12	11	6	9	4	3	2
Cum Plan	5	12	22	28	36	48	56	62	71	76	78	79
Cum Act	4	9	17	22	31	43	54	60	69	73	76	78

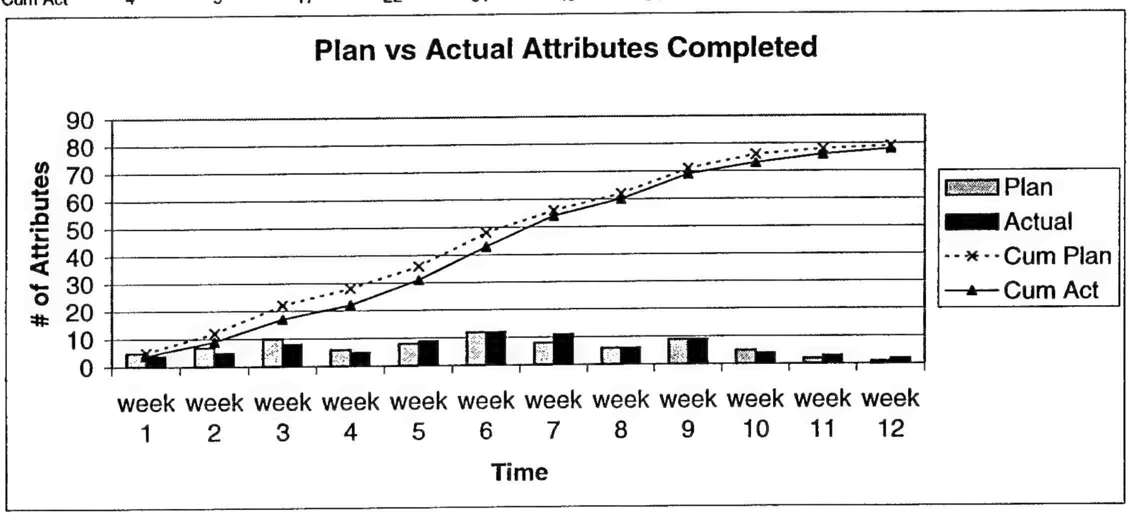


Figure 3.3-4. Plan versus actual attributes completed.

3.3.2.4 Class Traceability Status

Issue Category	Schedule and Progress Class Status
Measure	Number of classes per CRP Number of classes traced to use cases per CRP Number of derived classes per CRP Number of classes not yet traced to use cases per CRP Calculate cumulative number of classes defined (ref. App. A, (p)) Calculate cumulative number of classes traced to use cases or derived (ref. App. A, (p))
Description	The class traceability status metric measures the degree to which software design products have implemented the software requirements.
Example Graph	A line chart combined with a bar chart (Figure 3.3-5) is used to present the traceability information. The information presented includes a cumulative traceability status. This is augmented by a bar chart indication of the CRP use case traceability status.
Performance Analysis	Traceability can be a valuable management support tool at system requirements, design or other joint reviews. It may also indicate those areas of software requirements or design which have not been properly defined. Persistent numbers of untraced classes can indicate problems in the requirements (use cases), design (classes) or both.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# SW Classes	1	3	6	3	7	8	9	5	7	3	2	1
Traced to Use Case	0	0	0	0	4	5	3	2	1	1	1	0
Derived Class	0	0	0	0	4	7	10	6	6	3	1	1
Not Traced (TBD)	1	3	6	3	7	8	9	5	7	3	0	0
Cum SW Classes	1	4	10	13	20	28	37	42	49	52	54	55
Cum SW Class Traced	0	0	0	0	8	20	33	41	48	52	54	55

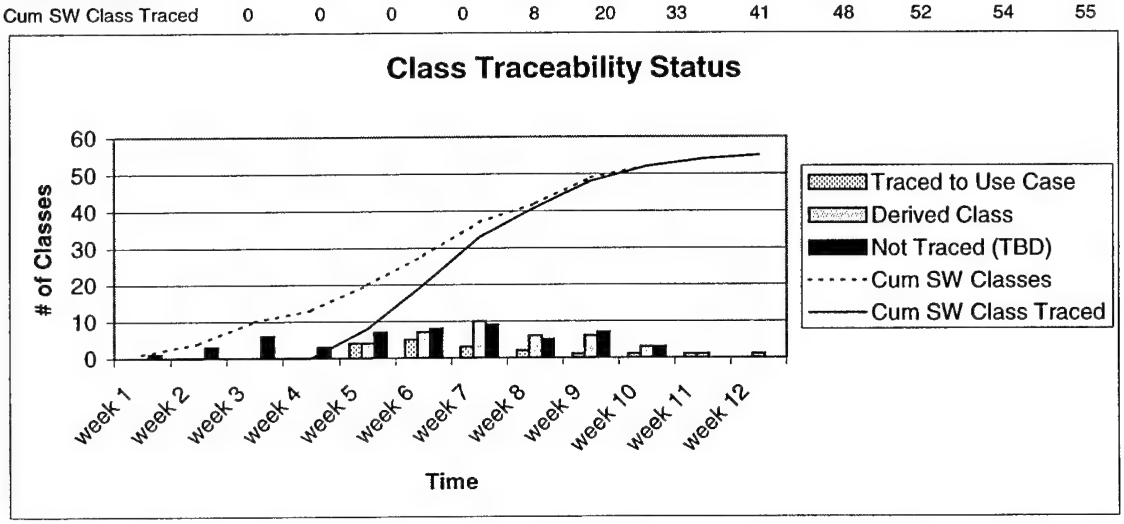


Figure 3.3-5. Class traceability status.

3.3.2.5 Integration Test Traceability Status

Issue	Schedule and Progress
Category	Class Status
Measure	Number of classes per CRP Number of classes traced to test cases per CRP Number of classes not yet traced to a test case per CRP Calculate cumulative number of classes defined (ref. App. A, (q)) Calculate cumulative number of classes traced to test cases (ref. App. A, (q))
Description	The integration test traceability status metric measures the degree to which the integration test cases cover the software design. The integration testing is focused on the structure of the system and primarily focuses on interface testing. Having traceability at this level ensures that appropriate structural coverage has been achieved in the testing.
Example Graph	A line chart combined with a bar chart (Figure 3.3-6) is used to present the traceability information. The information presented includes a cumulative traceability status. This is augmented by a bar chart indication of the CRP use case traceability status.
Performance Analysis	Traceability of classes to integration test cases provides the capability to track failed test cases back to areas of the software design. Persistent numbers of classes untraced to integration test cases can indicate an inadequate integration test program. This can lead to defects remaining in the software products until late in the program when it is more time consuming and expensive to fix them.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# SW Classes	1	3	6	3	7	8	9	5	7	3	2	1
Traced to Test Case	0	0	4	3	6	5	8	10	5	5	7	2
Not Traced (TBD)	1	3	6	3	7	8	9	5	7	3	0	0
Cum SW Classes	1	4	10	13	20	28	37	42	49	52	54	55
Cum SW Class Traced	0	0	4	7	13	18	26	36	41	46	53	55

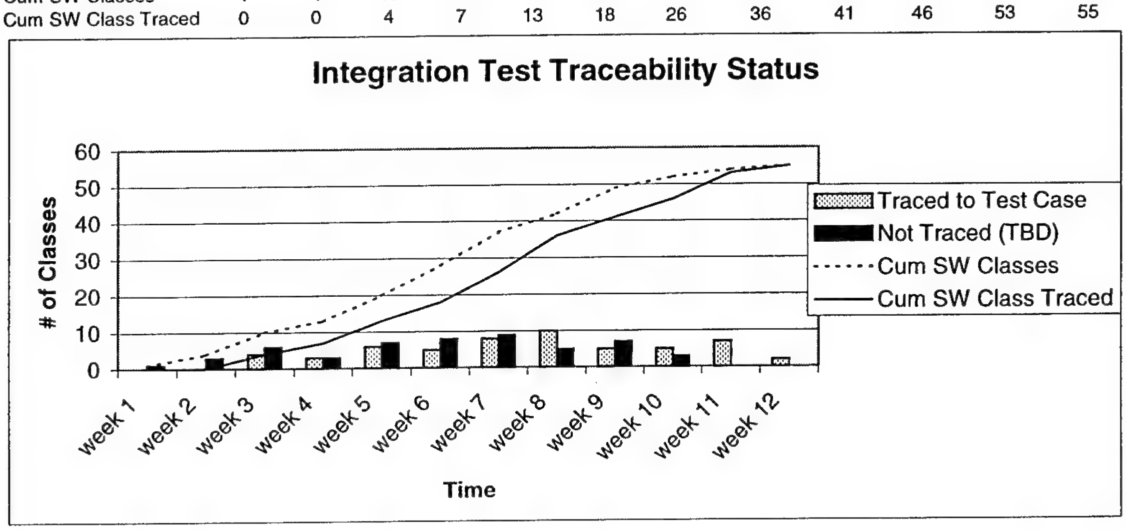


Figure 3.3-6. Integration test traceability status.

3.3.2.6 Plan verses Actual Classes that have Successfully Passed Integration Test

Issue	Schedule and Progress
Category	Class Status
Measure	Planned number of classes tested and passed per CRP Actual number of classes tested and passed per CRP Calculate cumulative number of planned classes tested and passed (ref. App. A, (s)) Calculate cumulative number of actual classes tested and passed (ref. App. A, (s))
Description	The Test Passed indicator monitors test progress during the integration and test phase of a software project. The criteria for determining whether a class has been successfully tested must be well defined for this metric to be meaningful.
Example Graph	A line chart combined with a bar chart (Figure 3.3-7) is used to present the test information. The information presented includes the planned number of successful test cases and the actual number of successful test cases plotted over time. A cumulative value is included in the line chart and a CRP plan/actual is provided in the bar chart.
Performance Analysis	If the actual number of classes successfully tested slips behind the number planned, this can indicate problems in the software integration test program itself (e.g., insufficient or inexperienced staff) or could indicate problems with the software itself (e.g. large numbers of defects identified by the integration testing).

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	3	5	7	4	6	10	6	4	7	3	1	1
Actual	1	3	6	3	7	8	9	5	7	3	2	1
Cum Plan	3	8	15	19	25	35	41	45	52	55	56	57
Cum Act	1	4	10	13	20	28	37	42	49	52	54	55

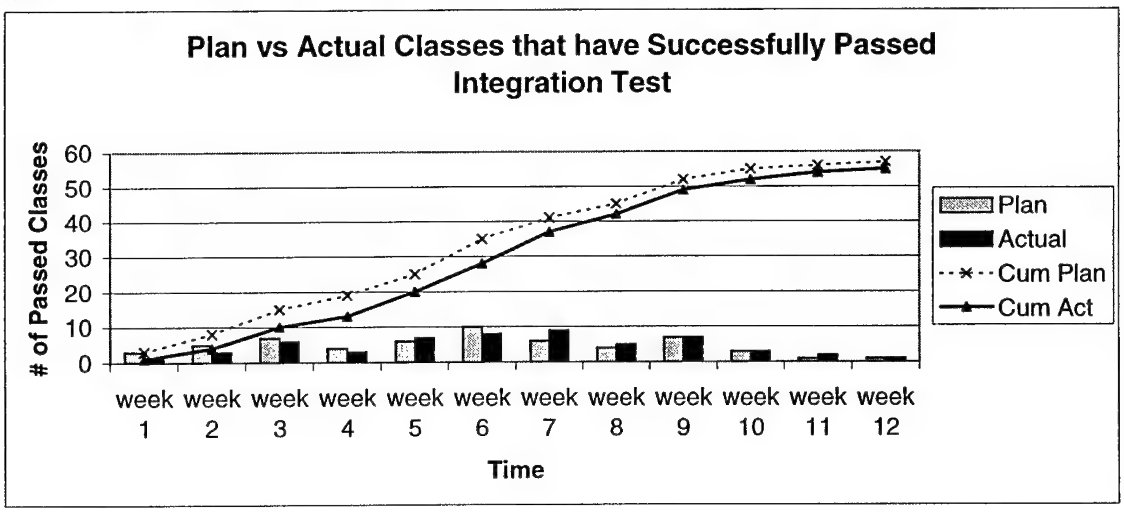


Figure 3.3-7. Plan verses actual classes that have successfully passed integration test.

3.3.3 Use Case Status

Use case status measures address progress based on the completion of work units that combine incrementally to form a complete software activity or product. If objective completion criteria are defined, use case status measures are extremely effective for assessing progress at any point in the project. Objective completion criteria are defined as "measurable and useful indicators that demonstrate that the achievement or maturity/progress in an activity or accomplishment has been achieved...Accomplishment criteria include, but are not limited to: 1) completed work efforts; 2) activities to confirm success of meeting technical, schedule, or cost parameters; 3) internal documents, which provide results of incremental verification; and 4) completion of critical process activities and products." [IMP/IMS] They are used for projecting completion dates for the activity or product.

3.3.3.1 Plan versus Actual Use Cases Completed

Issue	Schedule and Progress
Category	Use Case Status
Measure	Refer to section 3.1.1.1 (Issue: Growth and Stability, Category: Size).

3.3.3.2 Use Case Traceability Status

Issue	Schedule and Progress
Category	Use Case Status
Measure	Number of use cases per CRP Number of use cases traced to functional requirement(s) per CRP Number of derived use cases per CRP Number of use cases not yet traced to parent requirements per CRP Calculate cumulative number of use cases defined (ref. App. A, (o)) Calculate cumulative number of use cases traced to parent requirement(s) or derived (ref. App. A, (o))
Description	The traceability metric measures the degree to which software products have implemented functional requirements allocated from higher level specifications.
Example Graph	A line chart combined with a bar chart (Figure 3.3-8) is used to present the traceability information. The information presented includes a cumulative traceability status. This is augmented by a bar chart indication of the CRP use case traceability status.
Performance Analysis	Traceability can be a valuable management support tool at system requirement, design or other joint reviews. It may also indicate those areas of software requirements or design which have not been properly defined. Persistent numbers of untraced use cases can indicate problems in the software requirement definition (e.g. incompleteness, "gold plating", and incorrect software requirements).

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# SW Use Cases	1	3	6	3	7	8	9	5	7	3	2	1
Traced to High Reqt	0	0	0	0	4	5	3	2	1	1	1	0
Derived Use Cases	0	0	0	0	4	7	10	6	6	3	1	1
Not Traced (TBD)	1	3	6	3	7	8	9	5	7	3	0	0
Cum SW Use Cases	1	4	10	13	20	28	37	42	49	52	54	55
Cum SW UC Traced	0	0	0	0	8	20	33	41	48	52	54	55

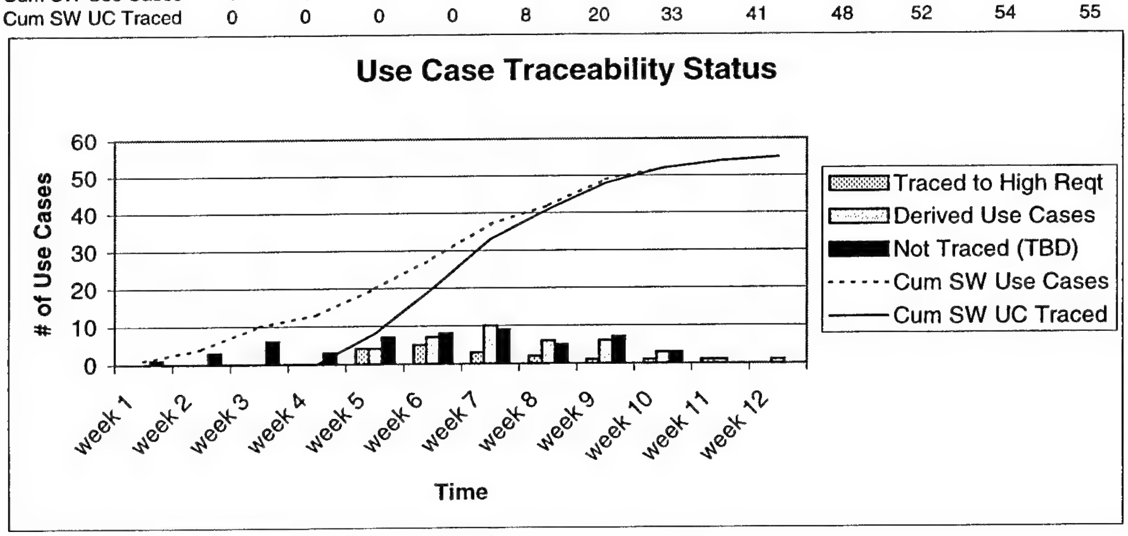


Figure 3.3-8. Use case traceability status.

3.3.3.3 Functional Test Traceability Status

Issue Category	Schedule and Progress Use Case Status
Measure	Number of use cases per CRP Number of use cases traced to test cases per CRP Number of use cases not yet traced to a test case per CRP Calculate cumulative number of use cases defined (ref. App. A, (r)) Calculate cumulative number of use cases traced to test cases (ref. App. A, (r))
Description	The traceability metric measures the degree to which the functional test cases cover the software use cases.
Example Graph	A line chart combined with a bar chart (Figure 3.3-9) is used to present the traceability information. The information presented includes a cumulative traceability status. This is augmented by a bar chart indication of the CRP use case traceability status.
Performance Analysis	Traceability of use cases to functional test cases provides the capability to track failed functional test cases back to the software requirements as defined by the use cases. The failed tests can thus be tracked back to specific mission needs. Persistent numbers of use cases untraced can indicate an inadequate software qualification test program. This can lead to defects remaining in the software product until later in the program when it is more time consuming and expensive to fix them.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
# SW Use Cases	1	3	6	3	7	8	9	5	7	3	2	1
Traced to Test Case	0	0	4	3	6	5	8	10	5	5	7	2
Not Traced (TBD)	1	3	6	3	7	8	9	5	7	3	0	0
Cum SW Use Cases	1	4	10	13	20	28	37	42	49	52	54	55
Cum SW UC Traced	0	0	4	7	13	18	26	36	41	46	53	55

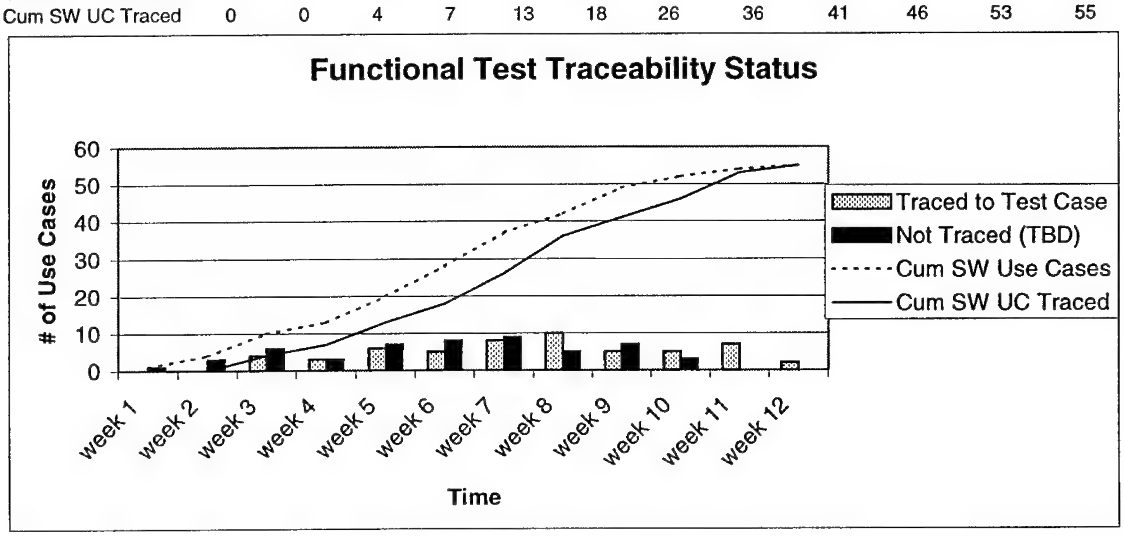


Figure 3.3-9. Functional test traceability status.

3.3.3.4 Plan versus Actual Use Cases that have Successfully Passed Functional Test

Issue Category	Schedule and Progress Use Case Status
Measure	Planned number of use cases tested and passed per CRP Actual number of use cases tested and passed per CRP Calculate cumulative number of planned software use cases tested and passed (ref. App. A, (t)) Calculate cumulative number of actual software use cases tested and passed (ref. App. A, (t))
Description	The Test Passed indicator monitors test progress during the qualification phase of a software project. The criteria for determining whether a use case has been successfully tested must be well defined for this metric to be meaningful.
Example Graph	A line chart combined with a bar chart (Figure 3.3-10) is used to present the test information. The information presented includes the planned number of test cases and the actual number of test cases plotted over time. A cumulate value is included in the line chart and a CRP plan/actual is provided in the par chart.
Performance Analysis	If the actual number of use cases successfully tested slips behind the number planned, this can indicate problems in the software qualification test program itself (e.g. insufficient or inexperienced staff) or could indicate problems with the software itself (e.g. large numbers of defects identified by the functional testing).

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Plan	3	5	7	4	6	10	6	4	7	3	1	1
Actual	1	3	6	3	7	8	9	5	7	3	2	1
Cum Plan	3	8	15	19	25	35	41	45	52	55	56	57
Cum Act	3	6	12	15	22	30	39	44	51	54	56	57

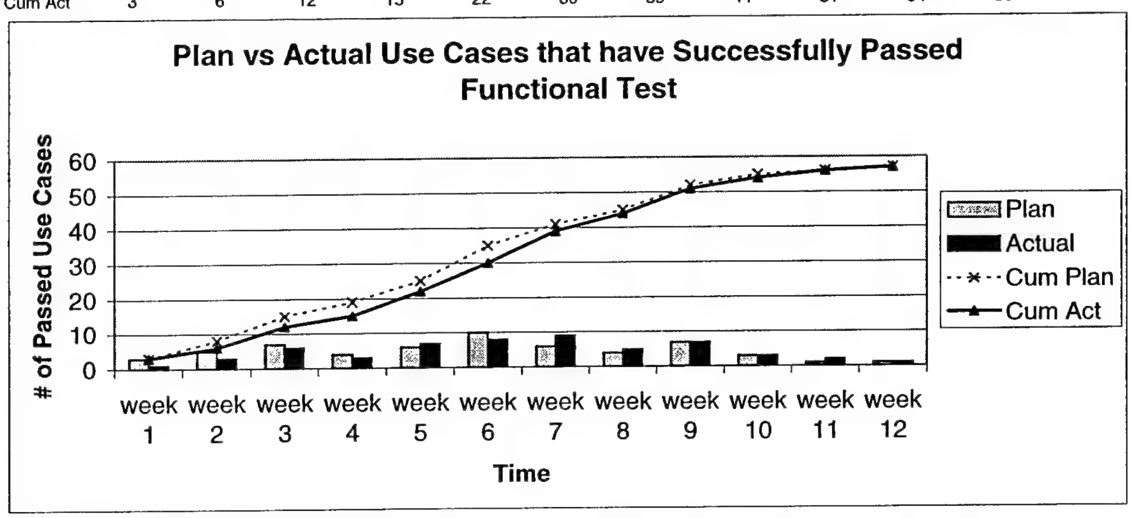


Figure 3.3-10. Plan verses actual use cases that have successfully passed functional test.

3.3.4 Build Content – Classes

These incremental capability measures count the classes associated with each incremental delivery. An incremental delivery may be a product shipped to a customer, or it may be an internal build delivered to the next phase of development. These measures are used to determine whether capability is being developed as scheduled or being delayed to future deliveries.

3.3.4.1 Plan versus Actual Classes in Build

Issue	Schedule and Progress
Category	Build Content - Classes
Measure	Planned number of classes in current build(s) per CRP Actual number of classes in current build(s) per CRP Cumulative number of planned classes in each build Cumulative number of actual classes in each build
Description	When multiple builds or releases are planned, this indicator helps determine if a realistic build schedule has been established and if progress in implementing the classes is tracking to the plan.
Example Graph	A line chart (Figure 3.3-11) is used to present the build information. The information presented includes a cumulative classes integrated into build status.
Performance Analysis	Deferments of classes to later builds without adjustments to the schedule are of greatest concern. A 5% or greater variance in a single build, or a 10% variance across two or more builds should be considered significant. During replans, this metric can assist in analyzing the updated build plans for feasibility based on past history.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Build 1 Plan	3	9	17	25								
Build 2 Plan					33	41	49	57				
Build 3 Plan									63	68	73	78
Build 1 Act	1	5	13	22								
Build 2 Act					33	44	55	66				
Build 3 Act									72	77	82	86

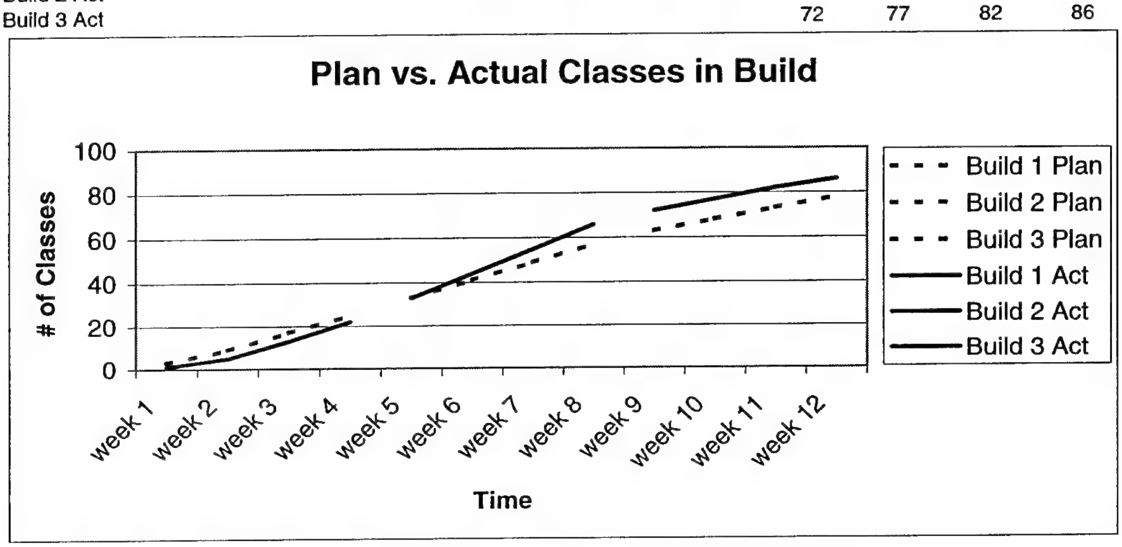


Figure 3.3-11. Plan versus actual classes in build.

3.3.4.2 Ratio of Classes in Build

Issue	Schedule and Progress
Category	Build Content - Classes
Measure	Planned number of classes in current build(s) per CRP Actual number of classes in current build(s) per CRP Calculate class integration slip ratio (ref. App. A, (v))
Description	This indicator helps identify the current status of the build by indicating whether fewer or more classes are implemented in the build than planned. Threshold Upper Bound +10% of plan. Threshold Lower Bound -10% of plan.
Example Graph	A line chart (Figure 3.3-12) is used to present the class integration slip information. The information presented includes a cumulative class integration slip value.
Performance Analysis	Slips in activities are of greatest concern due to the ripple effect in later parts of the schedule. This graph provides an overall assessment of the build. It is useful for ensuring that the project manager does not lose sight of the magnitude of the over-all schedule slippage. The ideal class integration slip ratio is 1.0 (plan = actual). Fewer classes implemented in the build than planned can indicate slippage of classes to later builds, with associated cost and schedule impacts. More classes implemented in the build than planned can indicate software design instability or uncontrolled addition of design features and capabilities.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Build 1 Plan	3	9	17	25	33	41	49	57	63	68	73	78
Build 1 Act	1	5	13	22	31	40	49	58	63	68	73	77
Integ Ratio	0.33	0.56	0.76	0.88	0.94	0.98	1.00	1.02	1.00	1.00	1.00	0.99
Upper Bound	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
Lower Bound	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9

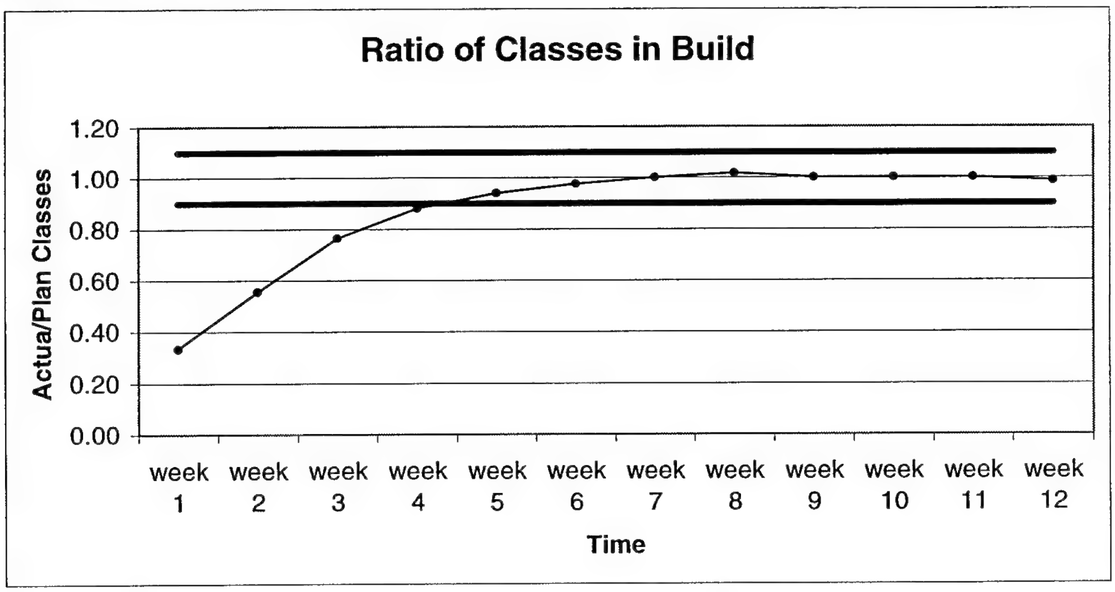


Figure 3.3-12. Ratio of classes in build.

3.3.5 Build Content – Use Cases

These incremental capability measures count the use cases associated with each incremental delivery. An incremental delivery may be a product shipped to a customer or it may be an internal build delivered to the next phase of development. These measures are used to determine whether capability is being developed as scheduled or being delayed to future deliveries.

3.3.5.1 Plan verses Actual Use Cases in Build

Issue	Schedule and Progress
Category	Build Content – Use Cases
Measure	Planned number of Use Cases in current build(s) per CRP Actual number of Use Cases in current build(s) per CRP Cumulative number of planned Use Cases in each build. Cumulative number of actual Use Cases in each build.
Description	When multiple builds or releases are planned, this indicator helps determine if a realistic build schedule has been established and if progress in implementing the software that performs the use cases is tracking to the plan.
Example Graph	A line chart (Figure 3.3-13) is used to present the build information. The information presented includes cumulative use cases integrated into build status.
Performance Analysis	Deferments of the use cases to later builds without adjustments to the schedule are of greatest concern. A 5% or greater variance in a single build, or a 10% variance across two or more builds should be considered significant. During replans, this metric can assist in analyzing the updated build plans for feasibility based on past history.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Build 1 Plan	3	8	15	19								
Build 2 Plan					25	35	41	45				
Build 3 Plan									52	55	56	57
Build 1 Act	1	4	10	18								
Build 2 Act					23	30	37	44				
Build 3 Act									49	51	53	55

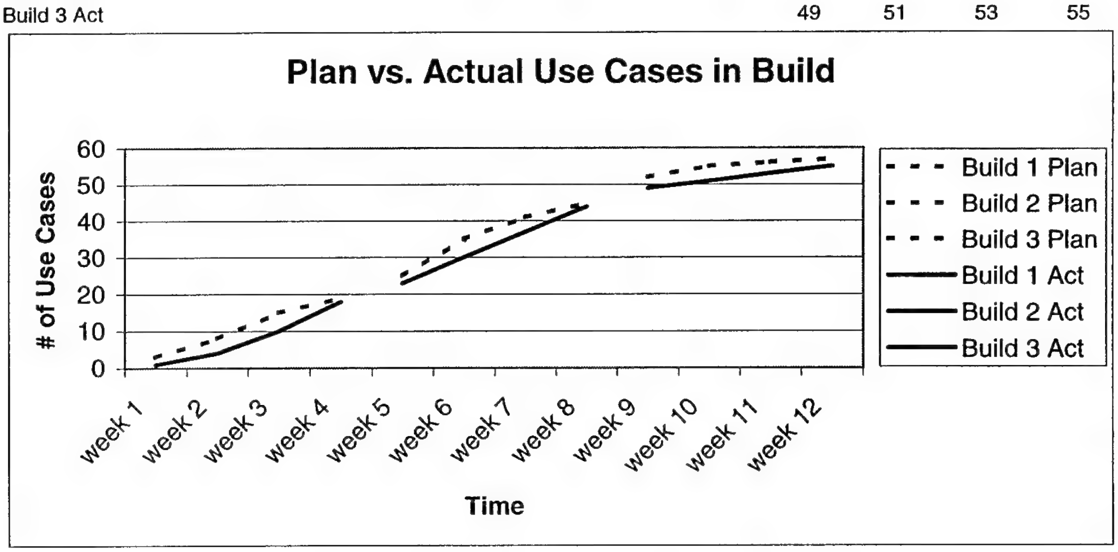


Figure 3.3-13. Plan verses actual use cases in build.

3.3.5.2 Ratio of Use Cases in Build

Issue	Schedule and Progress
Category	Build Content – Use Cases
Measure	Planned number of use cases in current build(s) per CRP Actual number of use cases in current build(s) per CRP Calculate use case integration slip ratio (ref. App. A, (u))
Description	This indicator helps identify the current status of the build by indicating whether fewer or more use cases are implemented in the build than planned. Threshold upper bound +10% of plan. Threshold lower bound –10% of plan.
Example Graph	A line chart (figure 3.3-14) is used to present the use case integration slip information. The information presented includes a cumulative use case integration slip value.
Performance Analysis	Slips in activities are of greatest concern, due to the ripple effect in later parts of the schedule. This graph provides an overall assessment of the build. It is useful for ensuring that the project manager does not lose sight of the magnitude of the overall schedule slippage. The ideal use case integration slip ratio is 1.0 (plan = actual). Fewer use cases implemented in the build than planned can indicate slippage of use cases to later builds, with associated cost and schedule impacts. More use cases implemented in the build than planned can indicate software requirements instability or uncontrolled software requirements growth.

	week 1	week 2	week 3	week 4	week 5	week 6	week 7	week 8	week 9	week 10	week 11	week 12
Build Plan	3	8	15	19	25	35	41	45	52	55	56	57
Build Act	1	4	10	18	23	30	37	44	49	51	53	55
Integ Ratio	0.33	0.50	0.67	0.95	0.92	0.86	0.90	0.98	0.94	0.93	0.95	0.96
Upper Bound	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1
Lower Bound	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9	0.9

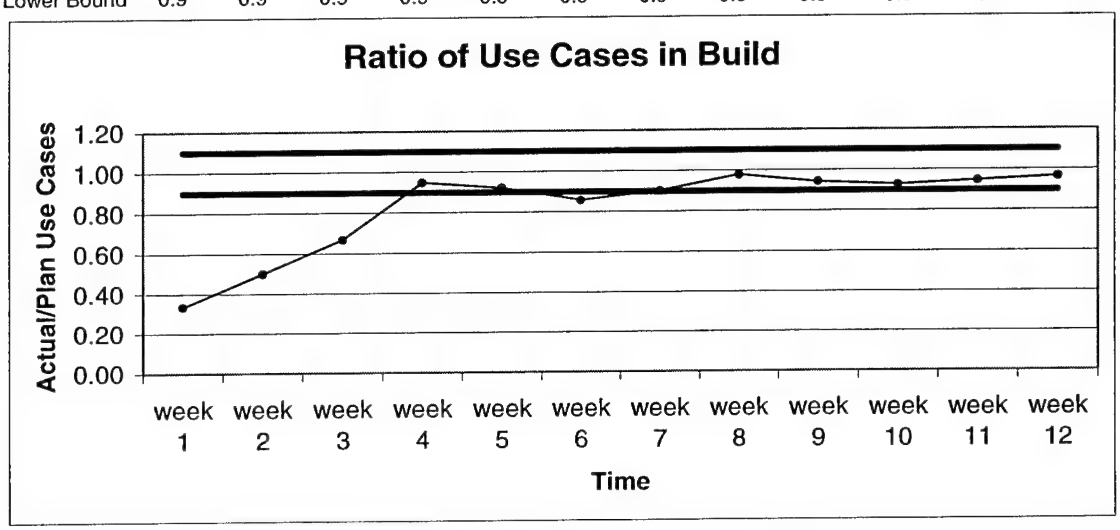


Figure 3.3-14. Ratio of use cases in build.

4. OO Technology Transition and Metrics Selection

Metrics selection for a program is difficult. Selecting metrics for use on an OO development program is especially challenging since OO techniques are currently in an industry-wide transition. The following guidance on metrics selection is provided to augment metrics selection information provided in more general materials such as Practical Software Measurement. [PSM] Specifically, this portion of the report addresses the unique problems associated with technology transitions and provides guidance on how to address metrics selection for an OO technology transition.

How well a developer uses OO technology depends on how far along that developer is on two curves. The first is the technology transition life-cycle curve, and the second is the project culture curve. Before making any specific OO metrics recommendations, acquisition personnel must assess where the developer is on these curves. That insight will help acquisition personnel understand why a developer is implementing the OO technology and OO metrics in a particular way and what recommendations would be effective. The developers themselves also need to understand their position along these two curves for effective transition to the OO technology. The following paragraphs describe the characteristics of these two curves. Following a description of the two curves, guidelines are provided on what metrics may be most useful for any given developer's location on these curves.

This section of the report is not intended to provide overall guidance on metrics selection, which is a much broader issue. The reader is referred to *Practical Software Measures* [PSM] for more information on the topic of how to select the full set of metrics to be used for a particular program.

4.1 Technology Transition Life cycle

The general model of the technology transition life cycle is shown in Figure 4.1-1. This model contains six phases: Contact, Awareness, Understanding, Trial Use, Limited Adoption, and

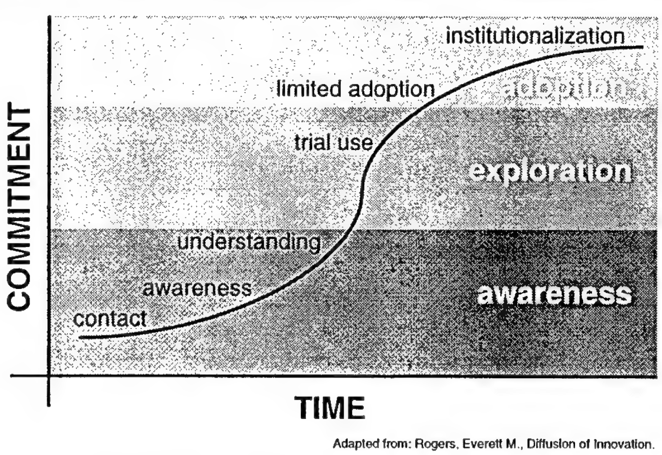


Figure 4.1-1. Technology transition life cycle.

Institutionalization. For the purposes of this description, these six phases will be summarized into three phases: Awareness, Exploration, and Adoption. In the awareness phase, the organization is becoming knowledgeable in the technology. In the exploration phase, the organization is becoming informed about how to apply the technology. And finally, in the adoption phase, the organization becomes adept in applying the technology.

The Awareness Phase is characterized by a low level of commitment (in terms of resources or people) and requires a long time to reach the next phase. During this phase, the organization is discovering what the technology is, developing a philosophy for using it, and beginning to understand what applications it can be applied to.

The Exploration Phase is characterized by a large increase in commitment over a short period of time. The organization at this phase has developed a use for the identified technology within the organization and has obtained a commitment to this idea from the decision-making structure of the organization. The decision-making structure has elected to utilize the technology, on a limited basis, to see whether there is merit to the idea. Being unprepared for the needed increased commitment in this phase can often be the death of a well-meaning technology transition program, while being successful during an exploration phase can catapult the technology into the mainstream of the organization. This phase is the make or break phase. Few technologies that are unsuccessful during an exploration phase are continued for further development.

The last and final phase is the Adoption Phase. During this phase, the organization has achieved some success in applying the technology in a limited sense, and the technology is transitioning to being applied globally throughout the organization. At this point, the technology will need to be able to be adaptable to various applications.

The technology transition curve relates to the DoD Acquisition life cycle in that one would expect to see mature technologies (adoption phase) being applied during the Production and Deployment phase. Less mature technologies (Exploration Phase) would be developed during the Concept & Technology Development phase. The System Development & Demonstration phase would be a mixture of the Adoption and Exploration Phase technologies. Technologies that are characterized as being in an awareness phase would be contracted for using the technology development contracts, which generally occur before a program enters the Acquisition life cycle.

Ideally, the software development technologies applied to a product development or system development and demonstration contract are all at the adoption phase since this would provide the least risk. However, this is not always possible, especially when multiple subcontractors are performing on a contract. In this case, for a given technology, each of the subcontractors may be at a different place on the technology transition life-cycle curve.

4.2 Technology Adoption Curve

The technology adoption curve characterizes the culture of the organization that is attempting to apply the new technology. While not all individuals in an organization must exhibit a strong desire to

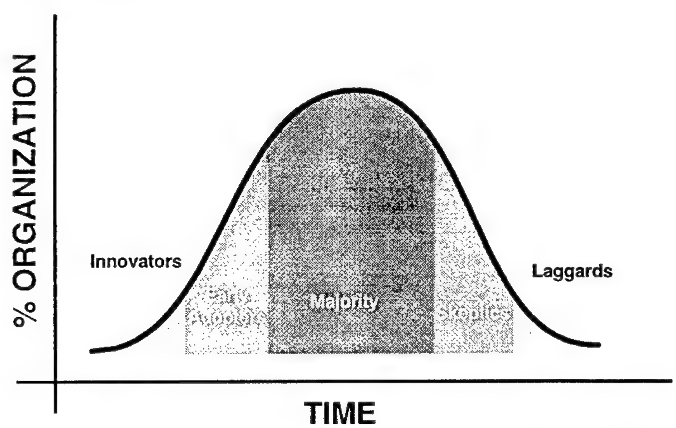
adopt the new technology, a majority of individuals must exhibit a willingness to at least accept the new technology.

The technology adoption curve shown in Figure 4.2-1 identifies the number of individuals that fall into various categories during a technology transition effort. This identification is shown as a function of time. Early on, very few people are involved and can be described as innovators. Later, more people become involved and can be described as early adopters. These people are now ready to manage the transition. Finally, the largest quantity of personnel (i.e., the majority) becomes involved in the adoption, but these individuals require proof of the effectiveness of the new technology before they can be full adopters. Finally, the heavy skeptics and laggards may become involved, but cannot be counted on to support the technology transition. A successful technology adoption would enlist the first three categories of individuals within an organization (i.e., innovators, early adopters, and the majority).

4.3 Relationship Between Technology Transition Life cycle and Technology Adoption Curve

The relationship of the technology transition life-cycle curve to the technology adoption bell curve is as follows. The innovators lead the effort performed during the awareness phase. During the Exploration Phase, a case is developed for why the technology is useful; additional individuals (i.e., the early adopters) become involved in determining the value of the technology to the organization. These people usually become technology managers. During this phase, some of the majority (i.e., the "need evidence") people are becoming involved, but it is still primarily driven by the innovators (i.e., "fanatics and champions") and the early adopters (e.g., "technology managers"). Finally, as the technology is rolled out in the organization, during the adoption phase, the majority of the organization becomes involved. During this phase, the skeptics and laggards have no choice but to become involved because the new technology has started to resemble a steam roller—adopt it or get run over.

Table 4.3-1 maps the major divisions of the technology transition life-cycle curve against the major divisions of the technology adoption curve.



Adapted from: Moore, Geoffry A., Crossing the Chasm.

Figure 4.2-1. Technology adoption curve.

Table 4.3-1. Technology Transition Life Cycle Divisions versus Technology Adoption Curve Divisions

CATEGORY	Technology Phase/ Staffed with	Awareness	Exploration	Adoption
	1. Innovators	Match	Match	Mismatch
	2. Early Adopters	Match	Match	Match
	3. Majority	Mismatch	Match	Match
	4. Skeptics	Mismatch	Mismatch	Match
	5. Laggards	Mismatch	Mismatch	Mismatch

Depending upon whether the characteristics of the technology adoption staff type is suited to the work being performed in the technology transition phase, each of the intersection cells in this matrix has been populated with either "match" or "mismatch." These two mappings identify when there is a match and mismatch between the technology transition life-cycle phase and the types of people from the technology adoption curve who perform that activity.

Consider, for example, the situation when any of the technology transition phases are staffed and led by a laggard who was highly successful on a previous program and has been selected for a leadership position on a newly won contract that is applying the latest technology currently being deployed in the R&D labs. This individual is well versed in the maintenance processes and has a heavy disposition towards installing the latest configuration management tools to ensure proper software control. However, the individual is not well versed in the technologies being applied on the new contract or in the project management implications of those technologies. This person's existing skills would not be applicable during the awareness or exploration phases. Only during the adoption phase, after the technology has been transitioned and is being successfully applied, would his skills be useful.

Metrics can be used to mitigate the risks of inadequate project management that can occur when there is a mismatch between the technology adoption staff type and the technology transition life-cycle phase. For example, if there is a laggard or an innovator performing the job of technology adoption and they are ill suited to perform the job, metrics can be used to compensate for their shortcoming. For instance, a laggard often is competent in project/program management and lacking in the technology. For this situation, stressing the technology product quality metric issue area will ensure that the technology is applied properly. Similarly, an early adopter often is competent in the technology but lacking in project/program management. For this situation, stressing the schedule and progress/growth and stability metrics categories will ensure that the project/program management effort is performed properly.

4.4 Selecting Object-Oriented Metrics

The technology transition life-cycle curve is used to gauge what metrics would be useful during the application of object-oriented technologies.

4.4.1 Technology Transition Life cycle – Awareness Phase

A program that is characterized as being in the Awareness Phase is one in which the organization adopting the technology has just barely scratched the surface of the technology. The project is probably struggling to determine how the technology will be used. The organization's technology awareness can be characterized as "one deep." There is probably just one person (or at most a few) who understands how the technology will be used and the methodology for using it. In this phase, a program should:

- Define and stabilize a process for using the technology
- Stabilize the product scope
- Have peer reviews of all technology artifacts with the technology "expert"

OO measurement categories that can help ensure that these things are happening are Use Case Status and Requirements Volatility. Other metrics categories useful for this purpose include Staff Experience, Staff Turnover, CMM Level, and Review Status. These measurements are summarized in Table 4.4-1.

Table 4.4-1. Goal Question Metric Applied to the Awareness Phase

Goal	Question	Category
Define and stabilize technology process	Is there organizational experience in applying new technology?	• CMM Level
Stabilize product scope	Are requirements activities on track? Have the requirements stabilized?	• Use Case Status • Requirements Volatility
"Expert" peer reviews	Who's on the project? How frequently do personnel change? Are the review activities on track?	• Staff Experience • Staff Turnover • Review Status

4.4.2 Technology Transition Life cycle – Exploration Phase

A program that is characterized as being in the Exploration Phase is usually one in which a number of individuals have become believers in the technology but do not fully understand it. At this step, most of the organization is in favor of the new technology, but, in many cases, the organization is using the terminology of the new technology to describe what has been done in the past. This is similar to what happened in the mid-1980's with the development of "Ada-tran" (FORTRAN constructs used in the Ada programming language) software. In the OO transition, for example, shared data items may be referred to as objects. In this phase, a program should:

- Stabilize the product design as soon as possible
- Define what quality criteria the OO artifacts and code will comply with

- Monitor quality criteria metrics regularly to ensure proper technology application
- Have peer reviews of all design decomposition artifacts with the technology "expert"

OO measurement categories that can help to ensure that these things are happening include Inheritance, Object Structure, Coupling, Class Status, and Size. Other measurement categories useful for this purpose include Staff Experience, Staff Turnover, and Review Status. These measurements are summarized in Table 4.4-2.

Table 4.4-2. Goal Question Metric Applied to the Exploration Phase

Goal	Question	Category
Stabilize product design	Are design activities on track?	• Class Status • Design Volatility
Define and monitor OO artifacts and code quality criteria	Have quality criteria been defined? Are quality criteria being monitored? Are OO artifacts and code quality within normal range?	• Inheritance • Object Structure • Coupling
"Expert" peer reviews	Who's on the project? How frequently do personnel change? Are the review activities on track?	• Staff Experience • Staff Turnover • Review Status

4.4.3 Technology Transition Life cycle – Adoption Phase

A program that is characterized as being in the Adoption Phase is usually one in which a cadre of individuals exist that have actually used the technology and know how to apply it. In this environment, the experienced individuals will not be dedicated to a specific program; they have become a corporate-wide resource. The emphasis should be on ensuring that each program becomes proficient in what the experts know as soon as possible. In this phase, a program should:

- Stabilize the product scope
- Stabilize the product design
- Define what quality criteria the OO artifacts and code will comply with
- Monitor quality criteria metrics regularly to ensure proper technology application
- Manage evolutionary builds

OO measurement categories that can help to ensure that these things are happening include Milestone Status, Build Content, Size, Requirements Volatility, Design Volatility, Inheritance, Object Structure, Coupling, Class Status, and UseCase Status. These measurements are summarized in Table 4.4-3.

Table 4.4-3. Goal Question Metric Applied to the Adoption Phase.

Goal	Question	Category
Stabilize product scope	Are requirements activities on track? Have the requirements stabilized?	• Use Case Status • Requirements Volatility
Stabilize product design	Are design activities on track?	• Class Status • Design Volatility
Define and monitor OO artifact and code quality criteria	Have quality criteria been defined? Are quality criteria being monitored? Are OO artifacts and code quality within normal range?	• Inheritance • Object Structure • Coupling
Manage Evolutionary Builds	Is the program on Track? Is functionality being slipped?	• Milestone Status • Build Content

5. Summary

The Object-Oriented methodology is the latest in the development philosophies to be transitioned into the DoD software development community. This methodology is particularly compatible with the evolutionary software development life-cycle model currently being required for use on all DoD large software-intensive system acquisitions. This life-cycle management approach is based on the evolutionary development philosophy where the software life cycle is iterated a number of times to develop multiple object-oriented products of increasing capability. These two components, the object-oriented methodology and its associated evolutionary life cycle, are referred to as object technology. The application of object technology to the development of software-intensive systems provides some unique management challenges.

Measurement is a critical tool in the quest for improved product quality and decreased cost and schedule. In fact, effective management and control of a large software system development effort are not possible without it. While many organizations use some form of software measurement, most do not have mature measurement programs. In addition, most organizations that have made or are making a transition to OO development have not adapted their measurement programs to fully address OO development products, processes, and resources. The guidance in this report is meant for organizations and projects that need to adapt their existing measurement programs for OO development. The intent of this report is to augment existing metrics information with a primer on the unique aspects of OO development and how object technology drives metrics selection and use.

A software development project utilizing Object technology has the same software development issues that a conventional project would. For this reason, this report is presented utilizing the format of the industry-accepted Practical Software Measures. [PSM] This framework is then extended to include the OO-unique issues.

For OO development,metrics can be helpful not only in providing information on development status, problems, and risks, but also in evaluating the effectiveness of OO methods and tools themselves. Measurement can be used to track progress toward taking full advantage of the OO paradigm. By providing data on OO structural attributes, OO metrics can help assess whether or not OO methods are being used as intended to facilitate modifiability and reusability. Finally, as organizations modify their OO processes with the intent of improvement, OO metrics are needed to assess whether or not intended improvements are indeed realized. Thirty-one detailed metrics descriptions specific to systems being developed with object technology are provided that cover the issues of growth and stability, product quality, and schedule and progress.

Initially a program selecting to use object technologies may require a technology transition phase in order to increase the understanding of developers, management, senior management, and customer personnel on the methodology and its application to the software life-cycle management techniques. Following this, a phase of institutionalization may be needed where the technique becomes part of normal operations. When these phases are complete, the organization will be using the object tech-

nologies effectively and efficiently as part of normal operations. One of the key areas in which management insight during a technology transition can be achieved is through the use of appropriate software development metrics for the OO paradigm. This report discusses how the technology transition lifecycle influences which types of metrics may be considered useful. For a full discussion of metrics selection reference the Practical Software Measurement [PSM].

Planning for the use of metrics during an acquisition is an important aspect of management oversight for both the acquirer and the developer. For the acquirer, a detailed description of what metrics information is needed and how to contractually transmit the information request to the developer has been provided in Appendix D.

For metrics use to be effective, it is necessary that the suppliers and users of metrics data understand the purpose of metrics use, what types of data can be collected, and how the data can be used. Projects and organizations that expect to benefit from metrics must be willing to commit significant resources to metrics implementation and use. They must be willing to take time to identify issues, risks, and information needs and to determine what can be measured to address them. In addition, they must be serious about developing a metrics program plan that precisely defines metrics and describes in detail the data collection, analysis, and interpretation processes.

Appendix A—Metrics Calculations

(a) Plan vs. Actual Use Cases Completed
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual

(b) Plan vs. Actual Classes Completed
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual

(c) Number of Attributes in a Class
Cum. Classes = Previous Week Cum. Classes + CRP # Classes
Cum. Attributes = Previous Week Cum. Attributes + CRP # Attributes
Ratio = CRP # Classes / CRP# Attributes
Cum. Ratio = Cum. Classes / Cum. Attributes

(d) Number of Methods in a Class
Cum. Classes = Previous Week Cum. Classes + CRP # Classes
Cum. Methods = Previous Week Cum. Methods + CRP # Methods
Ratio = CRP # Classes / CRP # Methods
Cum. Ratio = Cum. Classes / Cum. Methods

(e) Number of Scenarios in a Use Case
Cum. Use Cases = Previous Week Cum. Use Cases + CRP # Use Cases
Cum. Scenarios = Previous Week Cum. Scenarios + CRP # Scenarios
Ratio = CRP # Use Cases / CRP # Scenarios
Cum. Ratio = Cum. Use Cases / Cum. Scenarios

(f) Added, Deleted and Modified Use Cases
Actual = Base + CRP Added – CRP Deleted
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + Actual
Churn Ratio = (CRP Modified / Cum. Actual)

(g) Added, Deleted and Modified Classes
Actual = Base + CRP Added – CRP Deleted
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + Actual
Churn Ratio = (CRP Modified / Cum. Actual)

(h) Plan vs. Actual Methods Completed
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual

(i) Added, Deleted and Modified Methods
Actual = Base + CRP Added – CRP Deleted
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual
Churn Ratio = (CRP Modified / Cum. Actual)

(j) Plan vs. Actual Attributes Completed
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual

(k) Added, Deleted and Modified Attributes
Actual = Base + CRP Added – CRP Deleted
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual
Churn Ratio = (CRP Modified / Cum. Actual)

(l) Type of Method in Class
Cum. Private = Previous Week Cum. Plan + CRP Plan
Cum. Protected = Previous Week Cum. Actual + CRP Actual
Cum. Public = Previous Week Cum. Actual + CRP Actual

(m) Plan vs. Actual Milestone
Total Variance = (Actual Days – Plan Days) + Days Late Starting
Cum. Variance = Previous Week Cum. Variance + Total Variance

(n) Milestone Slip Ratio
Variance = (Actual Days – Plan Days) + Days Late Starting
Slip Ratio = Variance / Plan Days

(o) Use Case Traceability Status
Cum. Use Cases = Previous Week Cum. Use Cases + CRP # Use Cases Defined
Cum. Use Cases (UC) Traced = Previous Week Cum. Use Cases Traced + CRP Use Cases Traced to a Higher Requirement + CRP Derived Requirements

(p) Class Traceability Status
Cum. Classes = Previous Week Cum. Classes + CRP # Classes Defined
Cum. Classes Traced = Previous Week Cum. Classes Traced + CRP Classes Traced to Use Cases + Derived Class

(q) Integration Test Traceability Status
Cum. Classes = Previous Week Cum. Classes + CRP # Classes Defined
Cum. Classes Traced = Previous Week Cum. Classes + CRP Classes Traced to Integration Test Cases

(r) Functional Test Traceability Status
Cum. Use Cases = Previous Week Cum. Use Cases + CRP # Use Cases Defined
Cum. Use Cases Traced = Previous Week Cum. Use Cases Traced + CRP Use Cases Functional Test Cases + Derived Requirements

(s) Plan vs. Actual Classes that have Successfully Passed Integration Test
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual

(t) Plan vs. Actual Use Cases that have Successfully Passed Functional Test
Cum. Plan = Previous Week Cum. Plan + CRP Plan
Cum. Actual = Previous Week Cum. Actual + CRP Actual

(u) Ratio of Use Cases in Build
CRP Use Case Integ. Slip Ratio = CRP Build Actual Use Cases / CRP Build Plan Use Cases

(v) Ratio of Classes in Build
CRP Class Integ. Slip Ratio = CRP Build Actual Classes / CRP Build Plan Classes

Appendix B—Referenced Documents

BOOCH	Booch, Grady. *Object-Oriented Analysis and Design with Applications.* Benjamin/Cummings Publishing. Redwood City, CA. 1994.
BRIAND	Briand, Lionel C., Daly, John W. and Jürgen K. Wüst. "A unified framework for coupling measurement in object-oriented systems," *IEEE Transactions on Software Engineering*, vol. 25, no. 1. January/February 1999.
CHIDAMBER	Chidamber, Shyam R. and Chris F. Kemerer. "A metrics suite for object-oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6. June 1994.
COSTELLO	Costello, Rita J. and Sharon K. Hoting. "Metrics for Software-Intensive Mission Critical Computer Resource Systems, Version 1." Aerospace Report No. TOR-96(8617)-1. September 1996.
DEMARCO	DeMarco, Tom. *Structured Analysis and System Specification.* Prentice-Hall, Inc., Englewood Cliffs, New Jersery. 1979.
GSAM	*Guidelines for Successful Acquisition Management of Software Intensive Systems, Version 3.0.* Department of the Air Force: Software Technology Support Center. 2000.
HOTING	Hoting, Sharon K. and Rita J. Costello, "Computer Systems Division Software System Metrics Approach Revision 1." Aerospace Technical Report TR-96(8617)-1. September 1996.
IMP/IMS	Center Preparation Handbook Integrated Master Plan (IMP) Integrated Master Schedule (IMS), Aeronautical Systems Center, Wright-Patterson Air Force Base, Ohio. 17 August 1995.
JACOBSON	Jacobson, Ivar, Booch, Grady, and James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley. Reading, Massachusetts. 1999.
JENSEN	Jensen, Randall W., "Don't Forget About Good Management." *CROSSTALK: The Journal of Defense Software Engineering*, vol. 13, no. 8. August 2000.
KRUCHTEN1	Kruchten, Phillippe. "From Waterfall to Iterative Lifecycle – A tough transition for project mangers." Rational Software White Paper. http://www.rational.com/products/whitepapers/102016.jsp.
KRUCHTEN2	Kruchten, Phillippe. *The Rational Unified Process: An Introduction.* Addison-Wesley Longman, Inc., 1999.
LORENZ	Lorenz, Mark and Jeff Kidd. *Object Oriented Software Metrics.* PTR Prentice Hall, Englewood Cliffs, New Jersey. 1994.

MCCABE McCabe and Associates Website. http://www.mccabe.com/products/software_metrics.htm

MOORE Moore, Geoffry A., *Crossing the Chasm*, Harper Business, New York, NY. 1991.

OMG *OMG Unified Modeling Language Specification, Version 1.3.* Object Management Group, Inc. June 1999.

PSM "Practical Software Measurement," Office of the Under Secretary of Defense for Acquisition and Technology Joint Logistics Commanders Joint Group on Systems Engineering (OUSD A&T JGSE), Version 3.1a, April 1998.

RATIONAL "Rational Unified Process: Best Practices for Software Development Teams." Rational Software White Paper. http://www.rational.com/products/whitepapers/100420.jsp.

ROGERS Rogers, Everett M., *Diffusion of Innovation*. Free Press, New York, NY. 1983.

ROSENBERG Rosenberg, Linda H. and Lawrence E. Hyatt. "Software quality metrics for object-oriented environments". *CROSSTALK: The Journal of Defense Software Engineering,* vol. 10, no. 4. April 1997.

SPMN *Program Mangers Guide to Software Acquisition Best Practices, Version 2.31.* Computers & Concepts Associates in the performance of Space and Naval Warfare Systems Command (SPAWAR) Contract Number N00039-94-C-0153 for the operation of the Software Program Managers Network (SPMN). 1988.

TROUP Troup, Bonnie R. and Brian P. Gallagher. "Using contractor capability evaluations to reduce software development risk". *CROSSTALK: The Journal of Defense Software Engineering*, vol. 12, no. 8. August 1999.

YOURDON Yourdon, Edward and Larry L. Constantine. *Structure Design Fundametnals of a Discipline of Computer Program and System Design.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey. 1978.

Appendix C—OO Technique Primer

C.1 The Evolution of Software Development.

Early computers were strong boxes with weak brains. These early computers were expected to perform small, specific tasks. A programmer's job was to write code that performed that small task. It took little, if any, analysis to figure out how to solve the problem. As computer capacity grew, the number of tasks a computer was required to perform increased, but each individual task remained relatively simple.

Programming then evolved from a task orientation to a system orientation. The software system integrates many individual tasks into a single application. The integrated tasks are required to share data, communicate, and otherwise automate the operations that were previously run manually one piece at a time. Programming systems is more difficult than programming tasks. Programmers did not have the experience or tools to think about systems. Furthermore, the complexity of the software surpassed the analysis abilities of most software developers.

Flowcharts were the first widely used analysis tool. A flow chart depicts the functional flow of a system. The process of flowcharting allows developers to think about and solve problems on paper before writing code. As an added benefit, flow charts are useful to the people tasked with maintaining systems.

Structured programming in higher order languages (HOL) replaced flowcharts as the design tool of choice. Early software engineering researchers convinced programmers that removing line numbers, lining up an "if" with its "endif", and eliminating "goto" statements would eradicate bad buggy software. Structured programming helped a little; however, bad software continued to be produced.

Next came the structured analysis (SA) techniques, and many tool vendors supplied design decomposition tools to provide a structured mechanism to document the system to the level at which code could be developed. The thought was that because a coherent cohesive design had been developed from the "top down," there would be a high probability that the code could integrate successfully from the "bottom up." The structured analysis movement led to an integration philosophy of either "top down" or "bottom up." During the structured analysis era, the integration philosophy was influenced by the design decomposition methodology.

The state of the practice has recently evolved to object orientation. This design mechanism focuses on data rather than functions and produces models that represent a "real world" view of the problem. Object orientation provides a more intuitive approach to decomposition and a less rigid approach to integration. Integration in the OO paradigm is principally thread or function based. This approach is feasible because the designs are based on data integration.

As with any adaptive system, the future evolutions fix some of the inadequacies in prior implementations. Object technology is no different. Specifically, object technology was designed to address the inadequacies experienced in developing large software systems. These systems are characterized as applications that exhibit a rich set of behaviors. Some characteristics of large software systems include:

- Complex problem domain
- Difficulty managing the development process
- Required system flexibility
- Difficulty characterizing the system

Failure to manage the complexity of large software systems results in projects that are late, over budget, and deficient in meeting their stated requirements.

Object technology was targeted to address the known shortcomings inherent in the development of large software systems. Due to the risks in developing large complex systems, it became imperative to implement object technology in order to mitigate their risks. Since complexity in large software systems will not decrease, software-developing organizations must learn to adapt to it by imposing a rigorous design approach.

There are two distinct aspects of applying object-oriented technologies. The first is a *methodology* for defining the design. The second is a philosophy for managing the *life cycle* of the project. Each of these will be discussed in the subsequent sections.

C.2 Comparison of the OO Methodology to the SA Method

A software development methodology can be simply defined as a set of procedures that are followed from the beginning to the completion of the software development process. Since the 1970s, there has been a proliferation of unique software design methodologies developed to solve different types of application problems. Even though these design methodologies have evolved to respond to a specific application domain, there are many common characteristics. The common characteristics include: (1) a mechanism for the translation of information domain representation into design representation, (2) a notation for representing functional components and their interfaces, (3) heuristics for refinement and partitioning, and (4) guidelines for quality assessments. These characteristics will be used to contrast OO and SA methodologies in the following sections.

The Structured Analysis (SA) methodology is a dataflow-oriented design approach. SA utilizes a technical graphics "language" and a set of procedures and management guidelines to implement the language. This language is called the language of Structured Analysis. The procedures for the SA language are similar to the guidelines used for engineering blueprint systems. Each SA diagram is drawn on a single page and contains three to six nodes with interconnecting arcs. There are two kinds of SA diagrams – the activity diagram and the data diagram. [DEMARCO]

The SA methodology provides a precise and concise representation scheme and a set of techniques to graphically define complex system requirements. Figure C.2-1 provides a representation of the notation used for the SA methodology. There is top-down decomposition with clear decomposition for input, output, and control mechanisms for each node. It is beneficial to segregate the data and the activities into two diagrams so that the diagrams are not cluttered. The notation also distinguishes between control data and input data. However, in systems where many hierarchical diagrams are involved, the additional control information on the diagrams can make it difficult to understand. [YOURDON]

SA offers two quality criteria for evaluating software design, one for the software system level and one for the module level. The software system-level criterion measures the connections to other modules (coupling), and the module-level criterion measures the intra-module unity (cohesion). Coupling provides a way of evaluating the inter-dependencies between modules. Since modules are the building blocks of a software system, their relationships will determine how well the system can be maintained or changed. If the modules are highly interdependent on one another, it will be more difficult to make changes to one module without affecting the others. Conversely, if the modules are highly independent from one another, it will be easy to maintain, and changes can be made on one module without affecting the others. Cohesion provides a way of evaluating the functional connections among a module's processing elements. The most desirable cohesion is one where a module performs a single task with individual data elements. The least desirable cohesion is one where a module performs a few different tasks with unrelated data structures. A good system design will have strong cohesion and weak coupling.

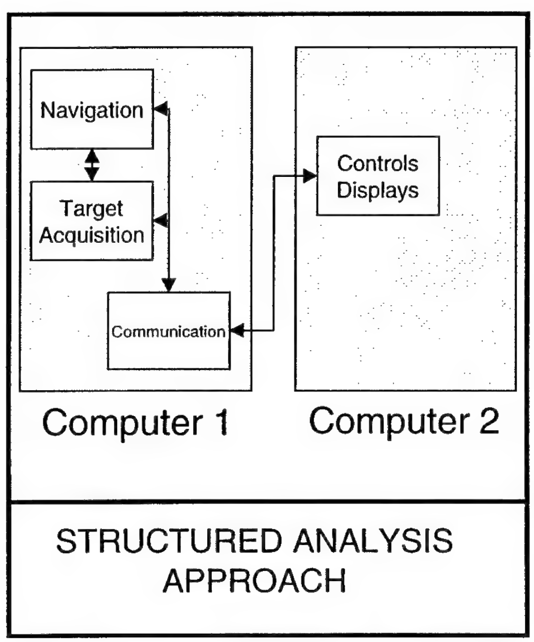


Figure C.2-1. Structured analysis model notation.

Object-Oriented Analysis (OOA) and Design (OOD) create a model of the real world and map it to the software structure. OO technique model the problem in terms of its data (i.e., objects and classes) and the operations performed on them. Objects represent concrete entities, which are instances of one or more classes. Objects encapsulate data attributes, which can be data structures or just attributes, and operations, which are procedures. Operations contain methods, which are program code that operates on the attributes. A class is a set of objects that share a set of common structure and behavior. A class represents a type, and an object is an instance of a class. The derivation of subclasses from a class is called inheritance. Relationships describe the dependencies between classes and objects.

OO requirements analysis and design start at the top-level of the software system by identifying the objects and classes, their relationships to other classes, their major attributes, their inheritance relationships. Next a class hierarchy is derived. The basic building blocks for OO are the mechanisms for depicting the data structure, specifying the operation and invoking the operation. Figure C.2-2 represents an example OO diagram.

A detailed design representation is obtained by extracting the objects that are available from each class and defining their relationships to each of the other objects. Identifying classes and objects creates data abstractions; modules are defined and structures for software are established by assigning operations to the data; and interfaces are described by developing a mechanism for using the objects.

Once the objects have been identified, the sets of operations that act on the objects are defined. There are basically three types of operations: those that manipulate data, those that perform computation, and those that monitor an object. Defining the object and its operations alone is not enough to derive the program structure. The interface that exists between the overall structure and objects must be identified and defined. All of these elements are integrated into a program-like construct.

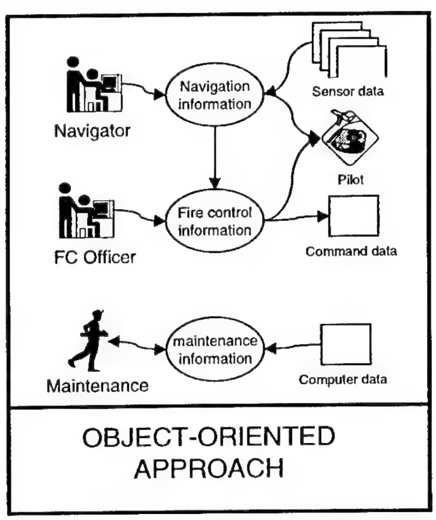


Figure C.2-2. Object-oriented model notation.

The Object-Oriented quality criteria for evaluating software design include five attributes: Coupling, Cohesion, Sufficiency, Completeness, and Primitiveness. Both coupling, the connections to other modules, and cohesion, the intra-module unity, have been borrowed from structured analysis. Closely related to the ideas of coupling and cohesion are the criteria that a class or module should be sufficient, complete, and primitive. Sufficiency means that the class or module captures enough characteristics of the abstraction to permit meaningful and efficient interaction. Completeness means that the interface of the class or module captures all of the meaningful characteristics of the abstraction. Where sufficiency implies a minimal interface, completeness implies the interface covers all aspects of the abstraction. Primitiveness applies to operations on data. An operation is primitive if additional functionality can be implemented only though access to the underlying representation. [BOOCH]

In summary, both SA and OO provide a means for generating a software design. A summary of the key points of each of the two methodologies is presented in Table C.2-1. This table is organized around the common methodology characteristics discussed at the beginning of this section.

Both the SA and OO methodology for developing software are well defined and have been proven useful in many domains. A comparison of these two methodologies is presented in Figure C.2-3. One of the features that OO provides that distinguishes it from SA is a focus on the real world. This approach makes the design models fully understandable by the domain experts. Another feature is the mapping of the real-world model to the software. This mapping is accomplished by domain-experienced engineers and provides full traceability between the "what" is needed and the "how" of implementation. Finally, the OO methodology provides a data centric orientation. This orientation is the most intuitive approach for many of the data centric systems under development today.

Table C.2-1. Comparison of Characteristics of the Structured Analysis and Object-Oriented Techniques.

	Structured Analysis	Object Oriented
Mechanism of Translation	Top down decomposition	Model of the "real world"
Notation	Activity diagrams Data diagrams	Objects Classes Relationships
Heuristics	Data Flow Control Flow	Modularity Abstraction Encapsulation
Quality Guidelines	Coupling Cohesion	Coupling Cohesion Sufficiency Completeness Primitiveness

• Creates models of processing system • Hierarchical • Processing Centric	• Creates map of real world • Real world maps to software • Data centric
STRUCTURED ANALYSIS APPROACH	OBJECT-ORIENTED APPROACH

Figure C.2-3. Comparison of structured analysis and object-oriented approaches.

One of the features of the SA approach that distinguishes it from the OO methodology is a precise and complete model of the computing system. This approach provides the programmers with a fully comprehensible model to implement. Another feature of the SA approach is that it provides a hierarchical model that hides unnecessary detail from the programmers until absolutely necessary. Finally, the SA approach details the processes that work on the data.

C.3 Contrast SA Life cycle to OO Life cycle

Software has been developed for the last two decades using the traditional SA approach or its variations. Each variation created a new set of problems for project management. With the emergence of object-orientated techniques there has been a radical shift in the development approach.

In parallel with the emergence of OO techniques, new approaches to the software life cycle have emerged. The software development life cycle can be characterized as either a "once-through" waterfall or iterative. There are currently several types of iterative life-cycle models in use, e.g., incremental, evolutionary, and spiral. Originally, SA was used with the waterfall or incremental models. The OO methodology is more suited to the newer evolutionary or spiral life cycles, although any of the life-cycle models can be used with either type of methodology. Figure C.3-1 summarizes the mapping of life-cycle models to methodology.

In the SA approach, software is developed using a top-down functional decomposition strategy, starting with a high-level view and progressively refining this view into a more detailed design. Implementation and integration are frequently accomplished using a "top down" or "bottom up"

	SA	OO
Waterfall	√	
Iterative		
Incremental	√	
Evolutionary	√	√
Spiral	√	√
OO Specific		√

Figure C.3-1. Life cycle to methodology mapping.

integration approach. The SA life-cycle activities comprise three main phases: analysis, design, and implementation, each executed a single time for any given iteration. Figure C.3-2 depicts the

waterfall or "once through" life-cycle model, which contains these three phases of activities in sequence. In the iterative life cycle models most applicable to SA (iterative, evolutionary, and some spiral instantiations), each iteration consists of the activities shown in Figure C.1.3-2 or of specific subsets of these activities.

In a waterfall approach, there is a lot of emphasis on "the specs" (i.e., the problem-space description) and getting them correct, complete, polished, and signed-off. In the iterative process, the software product takes precedence. The software architecture (i.e., the solution-space description) drives early life-cycle decisions. Customers do not buy specifications; it is the software product that is the main focus of attention throughout, with both specifications and software evolving in parallel. This focus on "software first" impacts the various teams. For example, in a waterfall development, testers may be used to receiving complete, stable specifications, well in advance of the start of testing. In an iterative development, the testers have to begin working at once on subsets of the software, with specifications and requirements that are still evolving. [KRUCHTEN1]

In contrast, since the OO-specific approach centers around modeling the "real world" in terms of objects that encapsulate data and operations, the OO life cycle supports the new programming phases of Inception, Elaboration, Construction, and Transition. During the Inception Phase, the "good idea" is defined, and the end-product vision is developed. The Inception Phase is concluded with the life-cycle objective (LCO) milestone. The Elaboration Phase consists of planning the necessary activities and required resources, specifying the requirements, and designing the architecture. The life-cycle architecture (LCA) milestone concludes the Elaboration Phase. The Construction Phase develops the product and evolves the vision until it is ready for delivery to its user community. The Initial Operation Capability (IOC) milestone concludes the Construction Phase. The Transition Phase provides for the transfer of the product to the users. This includes manufacturing, delivering, and training, and supporting and maintaining the product until the users are satisfied. The Transition Phase is concluded by the Final Operational Capability (FOC), which also concludes the life cycle. This phase relationship is depicted in Figure C.3-3.

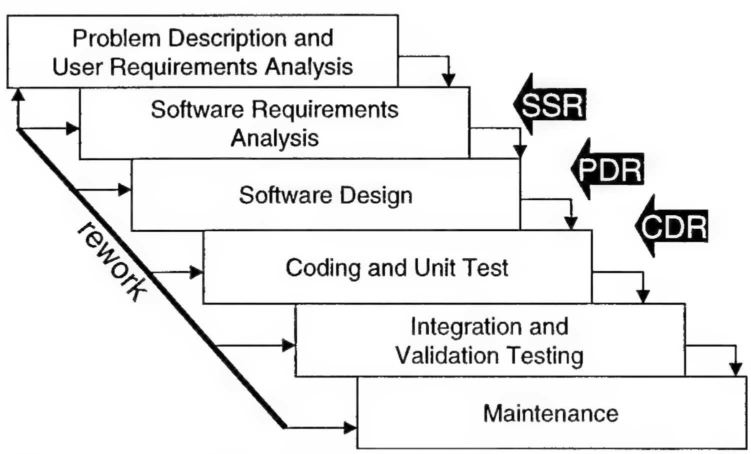
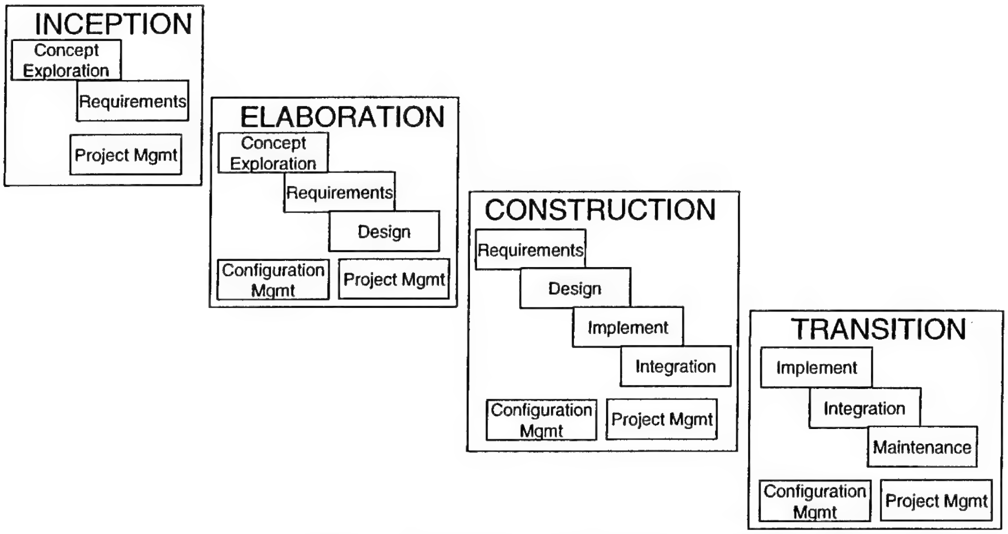


Figure C.3-2. Waterfall or "Once Through" software life cycle model.



Adapted from: The Rational Unified Process - An Introduction. Krutchten, Phillipe, Addison-Wesley, 1999.

Figure C.3-3. Iterative Development – shifting focus across the cycle.

In Figure C.3-3, each iteration follows a pattern similar to the waterfall approach, and, therefore, it contains the activities of requirements elicitation and analysis, of design and implementation, and of integration and test. However, from one iteration to the next and from one phase to the next, the emphasis on the various activities will change. Figure C.3-4 shows the relative emphasis of the various types of activities over time. In the Inception Phase the emphasis is mainly on understanding the overall requirements and determining the scope of the development effort. In the Elaboration Phase, the focus is on requirements, with some design and implementation aimed primarily at prototyping. During the Construction Phase the emphasis is on design and implementation. In the Transition Phase the focus is on ensuring that the system has the right level of quality to meet the objectives. In this way, the product evolves from initial conception to delivery through numerous iterations of requirements, design, code, integration, and maintenance. (An example of an OO-specific life-cycle model would be that which is described in the Rational Unified Process (RUP).) [RATIONAL]

CONS: • **"Big Bang" Integration Approach** • **Specification Based** **PROS:** • **Conceptually easy to understand** • **Simplified costing (reduced duplication)**	**CONS:** • **Difficult to Cost** • **Difficult to Manage** • **Difficult Accurate Accounting** **PROS:** • **Early risk mitigation** • **Better Change management** • **Potential exploitation of reuse** • **Lessons Learned rolled back into product => better quality**
WATERFALL APPROACH	**ITERATIVE APPROACH**

Figure C.3-4. Comparison of waterfall and iterative life cycle models.

In summary, both the Waterfall and Iterative life cycles for developing software are well defined and have been proven useful in many domains. A comparison of these two life cycles is presented in Figure C.3-4. Some of the benefits of the waterfall life cycle are that it is conceptually easy to understand and can be applied effectively to many problems. Another benefit of the waterfall approach is that it allows for a simple determination of how much it will cost to accomplish the job. There are several major benefits to the Iterative life cycles described above. These include (1) early risk mitigation, (2) better accommodation of changes, (3) potential for better exploitation of reuse, and (4) incorporation of lessons learned from previous iterations.

Appendix D—Acquisition and Development Metrics Planning

The acquirer and the developer both have significant roles to play in developing and using an effective metrics program. While metrics use is most often thought of as the act of collecting and analyzing data, software measurement encompasses a number of other activities as well, including:

- Selecting metrics.
- Developing a plan and process for implementing the metrics and for making changes to the metrics program.
- Collecting, calculating, analyzing, and reporting the metrics.
- Applying the metrics to identify and/or assess issues, risks, or problems.
- Using the results of these assessments to guide decisions and actions and track the effects of decisions and actions.

There are tasks for both the acquirer and developer in each of these activities.

D.1 The Acquirer's Role

The acquirer's role encompasses the following major tasks:

1. Determining the quantitative data (metrics) needed for the acquirer's program insight

   The acquirer should develop a list of known issues, risks, problems, and milestone/review/IPT events for the program's software effort and identify the types of data that will give relevant insight. Where possible (depending upon the acquisition phase and the degree of interaction between the acquirer and developer), the acquirer and developer should work together on this task.

2. Obtaining metrics planning information and metrics data from the developer

   The acquirer should ensure, through contractual means, that the developer is motivated to provide metrics plans and data to the acquirer. The preferred approach (acquisition strategy permitting) is to explicitly require the developer to deliver metrics plans and periodic metrics reports. Section D.3, Supplying Metrics Data, discusses this further. If such contractual requests are not allowed, and a capability evaluation [TROUP] is performed, it may be possible to obtain metrics-planning information in an SDP via appropriate tailoring of the evaluation instrument. If neither of these approaches applies to a program, it will be necessary to work directly with the developer to encourage metrics use and the sharing of metrics information.

3. Creating a plan for reviewing the developer's metrics plan and analyzing data

   The acquirer should assign responsibility for reviewing metrics plans and data, and ensuring that the assigned staff has the appropriate expertise to understand the information. Such expertise includes both software engineering and program knowledge. Training or briefings should be supplied at the executive and technical level as appropriate in preparation for analyzing and applying the data. The acquirer's plan should describe its analysis process for metrics data obtained from the contractor, including an approximate analysis schedule, the resources and skills required to perform the analysis, a list of the high-level steps in the analysis procedure, the format for reporting analysis results, and the action to be taken based on the information in the analysis reports.

4. Reviewing the developer's metrics program plan

   The developer and acquirer should work together as much as possible during development and maintenance of the metrics program plan. Where possible, the acquirer should have a major role in the metrics selection activity and at least a review role in the metrics definition activity. The acquirer should assign personnel with software engineering and program knowledge to evaluate the plan to ensure that it addresses key program objectives, issues, and risks, and that the selected metrics set covers all key software engineering processes, products, and resources. The adequacy of funding and personnel that the developer allocates to the metrics effort should also be assessed.

   As part of this task, the acquirer should document its assessment of the developer's metrics approach, metrics set, and metrics definitions, identifying any weaknesses or omissions. Even if this information is not supplied to, or used by, the developer, it will be useful to the acquisition personnel who must analyze the metrics data.

5. Analyzing the developer's metrics

   The acquirer should perform its own analysis of the developer's metrics data. First, an assessment of the validity of the data should be performed to ensure its currency, consistency, and accuracy. Then, the data should be analyzed and interpreted to identify and track the status of problems, risks, and issues. The analysis should also use the metrics data to assess the effectiveness of previous preventive and corrective actions.

6. Supplying feedback for needed changes to the metrics activity

   If shortcomings are discovered in the metrics set or the metrics program itself, or if new metrics are needed, the developer should be notified so that appropriate modifications can be made. Note that without sufficient contractual motivation, the developer will likely be reluctant to make modifications, especially if they appear to be costly or if the developer does not expect to benefit from them. Similarly, the acquirer may find deficiencies in its own metrics plan, and should modify it accordingly.

7. Ensure that metrics results are fed back into development so that corrections/adjustments can be made.

   All too frequently, metrics data is collected and reported but not actually used in managing the project. Both the developer and the acquirer must work to ensure that corrective actions are performed based on the analysis of the metrics data.

D.2 The Developer's Role

The developer has tasks to perform that parallel those of the acquirer. These are shown in Table D.2-1. At a minimum, the developer's tasks include preparing a metrics program plan in accordance with the instructions in the contract, allocating resources and providing any needed training to implement the plan, implementing the plan, performing all data collection, analysis, and reporting activities, and using the results for preventive and corrective action.

Table D.2-1. Metrics Program Planning Tasks

#	Acquirer	Developer
1	Determine metrics needed for acquisition insight	Determine metrics needed for development management
2	Request metrics planning information and metrics data from the developer	Create a metrics plan, data collection and analysis process, collection and reporting schedule and format for reporting to development management. Respond to the request by providing the acquirer with a metrics program information
3	Create a plan for reviewing the developer's metrics plan and analyzing metrics data	Incorporate the acquirer's review/analysis plans into the metrics program plan
4	Evaluate the developer's metrics program plan	Revise the metrics program plan, as appropriate
5	Analyze the developer's metrics data and provide analysis results to developer.	Implement the metrics plan, analyzing and using data in preventive and corrective action, using the data in process improvement, and supplying data to the acquirer for analysis Respond to analysis results from acquirer
6	Supply feedback to the developer on needed changes to the metrics activity	Analyze metrics program for needed improvements and improve metrics process accordingly Incorporate developer process improvement analysis and acquirer's feedback as appropriate

The developer's metrics program plan should address the topics in Table D.2-2. These topics were developed based on material from *Metrics for Software-Intensive Mission Critical Computer Resource Systems* [COSTELLO].

Table D.1-2. Topics for the Metrics Program Plan

Topics for the Metrics Program Plan
1. Introduction
2. Project and Organizational Context
Project technical characteristics
Project and organization management characteristics
Project and organization relevant past experience
Intended management use of the metrics information to assess and improve the software system product and the processes used to generate the product
Project-specific and organization-wide issues, risks, and information needs to be addressed with metrics
3. Roles and Responsibilities
Organization and individuals responsible for the metrics program overall
Roles and responsibilities for metrics selection, definition, planning, evaluation, and improvement
Roles and responsibilities for metrics collection, analysis, reporting, and feedback
Roles and responsibilities for metrics archival and configuration management
Measurement interfaces
Project software development organization to acquirer, user
Project software development organization to subcontractor(s)
Project software development organization to project software process group and/or overall development organization (the latter interface may be via the software process group)
Software-level metrics personnel to systems-level metrics personnel
Software metrics personnel to software QA, CM, safety, security, RMA, etc. personnel
Potential impediments to effective metrics use and how these will be addressed
4. General Approach
General procedures for collection, reporting, and data archival/configuration management
Tools for collection, reporting, and data archival/configuration management; and tool interfaces (automated or manual), where applicable
Overall aggregation structures applicable in data collection and "roll up"
5. Measurable Parameters
Products, processes, and resources relevant to the life cycle and their attributes to measure and to address identified problems, risks, issues, and information needs
Product, process, and resource characteristics needed to make these attributes measurable (e.g., structuring requirements so that they are individually countable
Interfaces to the system level for the software-related, system-level attributes to be measured
Interfaces to software quality and specialty engineering disciplines (e.g., CM, QA, safety, security, RMA)
Phased implementation of metrics, if applicable
6. Detailed Metrics Descriptions
Metrics Name
Metric Description
Primitive data elements – How defined, How and when collected
Equations for calculating the metric
Reporting Format – Graphical and numeric data
Analysis and interpretation guide
7. Metrics Program Evaluation and Improvement Process
Process for determining when improvements are needed
Process for updating metrics set/definitions while maintaining analysis capability on previous metrics set

D.3 Supplying Metrics Data

To enable the acquirer to make the most of the developer's metrics program, the developer should provide the acquirer with the following two types of information: (1) planning information that describes the metrics program in detail and (2) specific metrics data at regular reporting intervals. It cannot be assumed that the developer will supply the above information and reports to the acquirer, or even that the developer knows what constitutes good planning information or a complete metrics report. The latter might occur depending on where the developer is on the technology transition life-cycle curve and the technology adoption curve. Therefore, wherever possible, the acquirer must communicate its specific metrics needs and the developer's development management metrics needs to the developer in an explicit, contractually binding form.

D.3.1 Planning Information

Planning information consists of information on the overall approach and process (as described in Table D.1.-2) including complete descriptions of each metric

D.3.2 Metrics Reports

Specific metrics reports consist of graphical and numerical data, written analysis, and data in electronic form. Metrics reports should contain summary data, primitive data, and interpretive and contextual information. In general, high-level, summary data should be reported by the developer and discussed with the acquirer monthly. Primitive (low-level) data should be reported (or made available) monthly in electronic form to facilitate analysis by the acquirer. Interpretive and contextual information should be supplied as required in order to understand the metric report being presented.

Interpretive data should include graphical reports, numerical data supplements, and written analyses. Graphical reports must contain sufficient information so that they are not misleading. Numerical data supplements (e.g., in tabular form) may be helpful in this regard. Written analyses should be delivered with each report. These should explain the process for analyzing the metric; differences between the expected and actual values for the metric, if any; behavior of the metric in the context of other project information and metrics; and recommended actions based upon the analysis, if any.

LABORATORY OPERATIONS

The Aerospace Corporation functions as an "architect-engineer" for national security programs, specializing in advanced military space systems. The Corporation's Laboratory Operations supports the effective and timely development and operation of national security systems through scientific research and the application of advanced technology. Vital to the success of the Corporation is the technical staff's wide-ranging expertise and its ability to stay abreast of new technological developments and program support issues associated with rapidly evolving space systems. Contributing capabilities are provided by these individual organizations:

Electronics and Photonics Laboratory: Microelectronics, VLSI reliability, failure analysis, solid-state device physics, compound semiconductors, radiation effects, infrared and CCD detector devices, data storage and display technologies; lasers and electro-optics, solid state laser design, micro-optics, optical communications, and fiber optic sensors; atomic frequency standards, applied laser spectroscopy, laser chemistry, atmospheric propagation and beam control, LIDAR/LADAR remote sensing; solar cell and array testing and evaluation, battery electrochemistry, battery testing and evaluation.

Space Materials Laboratory: Evaluation and characterizations of new materials and processing techniques: metals, alloys, ceramics, polymers, thin films, and composites; development of advanced deposition processes; nondestructive evaluation, component failure analysis and reliability; structural mechanics, fracture mechanics, and stress corrosion; analysis and evaluation of materials at cryogenic and elevated temperatures; launch vehicle fluid mechanics, heat transfer and flight dynamics; aerothermodynamics; chemical and electric propulsion; environmental chemistry; combustion processes; space environment effects on materials, hardening and vulnerability assessment; contamination, thermal and structural control; lubrication and surface phenomena.

Space Science Applications Laboratory: Magnetospheric, auroral and cosmic ray physics, wave-particle interactions, magnetospheric plasma waves; atmospheric and ionospheric physics, density and composition of the upper atmosphere, remote sensing using atmospheric radiation; solar physics, infrared astronomy, infrared signature analysis; infrared surveillance, imaging, remote sensing, and hyperspectral imaging; effects of solar activity, magnetic storms and nuclear explosions on the Earth's atmosphere, ionosphere and magnetosphere; effects of electromagnetic and particulate radiations on space systems; space instrumentation, design fabrication and test; environmental chemistry, trace detection; atmospheric chemical reactions, atmospheric optics, light scattering, state-specific chemical reactions and radiative signatures of missile plumes.

Center for Microtechnology: Microelectromechanical systems (MEMS) for space applications; assessment of microtechnology space applications; laser micromachining; laser-surface physical and chemical interactions; micropropulsion; micro- and nanosatellite mission analysis; intelligent microinstruments for monitoring space and launch system environments.

Office of Spectral Applications: Multispectral and hyperspectral sensor development; data analysis and algorithm development; applications of multispectral and hyperspectral imagery to defense, civil space, commercial, and environmental missions.

THE AEROSPACE CORPORATION
2350 E. El Segundo Boulevard
El Segundo, California 90245-4691
U.S.A.